Modul Pelatnas IOAI Indonesia

Modul 3: Pengenalan Deep Learning

Tim Pembina IOAI Indonesia
sc.ioai.id@gmail.com

Juli 2025

Modul Pelatnas IOAI Indonesia

Modul 3: Pengenalan Deep Learning

Penyusun

Tim Pembina IOAI Indonesia

sc.ioai.id@gmail.com

Daftar Isi

Pengantar

Puji syukur kehadirat Tuhan YME atas berkat limpahan rahmat dan karnuia-Nya, Buku Modul Pelatnas IOAI ini telah berhasil kami selesaikan. Buku Modul ini kami susun sebagai salah satu referensi rangkaian pembinaan/pelatihan nasional bagi siswa peserta didik yang mengikuti Pelatnas dalam rangka membentuk tim yang akan mewakili Indonesia pada ajang International Olympiad in Artificial Intelligence (IOAI).

Terima kasih yang sebesar-besarnya kami sampaikan kepada para Pembina, Asisten Pembina, dan para Alumni ajang OSN bidang Informatika dan IOI (TOKI), serta semua pihak yang telah berkontribusi sehingga Buku Modul ini dapat terwujud. Kami menyadari masih banyak kekurangan dalam penulisan Buku Modul ini. Untuk itu kami mohon maaf dan kami sangat mengharapkan masukan untuk perbaikan dan penyempurnaan selanjutnya, sehingga keberadaan Buku Modul ini dapat memberikan manfaat yang sebesar-besarnya bagi semua pihak.

Semoga Buku Modul Pelatnas IOAI ini dapat digunakan sebaik-baiknya untuk menunjang kegiatan Pelatnas IOAI dan mampu membantu menghasilkan calon-calon talenta Indonesia di bidang AI yang mampu memberikan prestasi yang membanggakan di tingkat internasional.

BAB 1

Pengenalan Neural Network

Pendahuluan

Jaringan saraf tiruan (Artificial Neural Network, ANN) adalah salah satu pendekatan paling penting dalam bidang kecerdasan buatan dan pembelajaran mesin modern. Inspirasi awalnya berasal dari cara kerja otak manusia, di mana terdapat jaringan neuron biologis yang saling terhubung dan mampu belajar dari pengalaman. Walaupun sangat disederhanakan, jaringan saraf buatan mencoba meniru prinsip ini dengan menghubungkan banyak "unit" kecil (disebut neuron) yang melakukan perhitungan sederhana.

Perkembangan jaringan saraf telah membawa revolusi besar dalam berbagai bidang seperti pengenalan gambar, terjemahan bahasa, permainan, dan kendaraan otonom. Dalam bab ini, kita akan mempelajari dasar dari jaringan saraf, dimulai dari unit paling dasar yang disebut **perceptron**.

1.1 Konsep Jaringan Saraf Buatan

Sebuah jaringan saraf buatan tersusun dari banyak unit pemroses kecil yang terhubung, disebut *neuron*. Setiap neuron menerima satu atau lebih input, mengalikan masing-masing input dengan bobot (*weight*), menjumlahkannya, dan menerapkan fungsi aktivasi untuk menghasilkan output. Output ini kemudian dapat menjadi input bagi neuron lain.

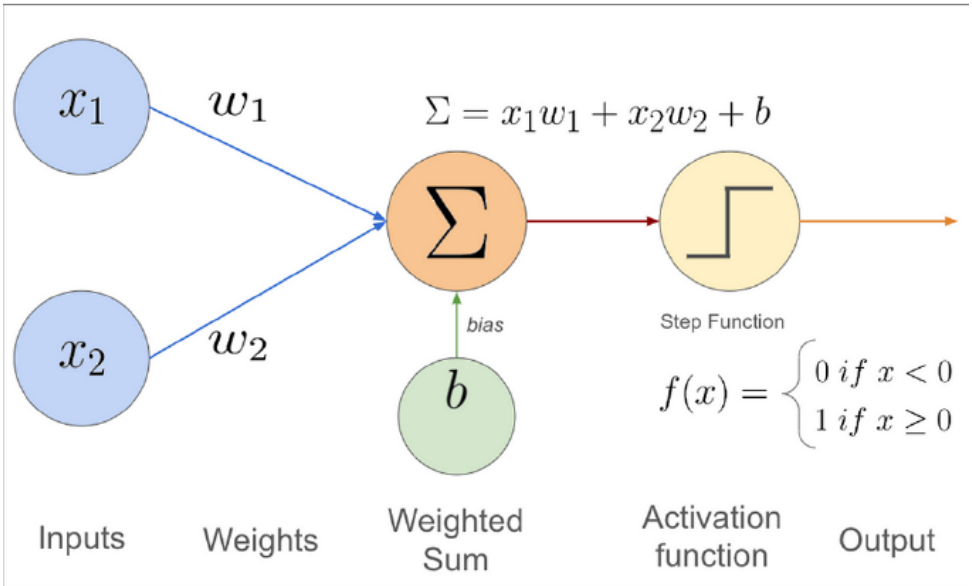
1.2 Perceptron: Unit Dasar Neural Network

1.2.1 Struktur Perceptron

Perceptron adalah jenis jaringan saraf paling sederhana. Ia terdiri dari:

- Satu atau lebih input: $x_1, x_2, ..., x_n$
- Bobot untuk setiap input: $w_1, w_2, ..., w_n$
- Bias b
- Fungsi aktivasi (misalnya step function)

Output dari perceptron adalah:



Gambar 1.1. Perceptron

$$y = \begin{cases} 1, & \text{jika } \sum w_i x_i + b \geq 0 \\ 0, & \text{jika tidak} \end{cases}$$

Gambar!1.1 menggambarkan sebuah perceptron

1.3 Melatih Perceptron

Agar sebuah perceptron dapat membuat prediksi yang sesuai dengan target, kita harus melatih bobot-bobotnya. Proses pelatihan ini dilakukan dengan menggunakan algoritma **Perceptron Learning Rule**, yang mengupdate bobot berdasarkan kesalahan prediksi.

Algoritma Pelatihan Perceptron

Diberikan input $\vec{x} = [x_1, x_2, ..., x_n]$ dan label target $y \in \{0, 1\}$, serta prediksi $\hat{y}$ dari perceptron, aturan update bobot adalah:

$$w_i \leftarrow w_i + \eta \, (y - \hat{y}) \, x_i$$
$$b \leftarrow b + \eta \, (y - \hat{y})$$

di mana:

- w_i: bobot ke-i
- b: bias
- η: *learning rate*
- $(y - \hat{y})$: error

Update ini dilakukan setiap kali prediksi salah. Proses ini diulang selama beberapa epoch hingga model mempelajari pola dari data.

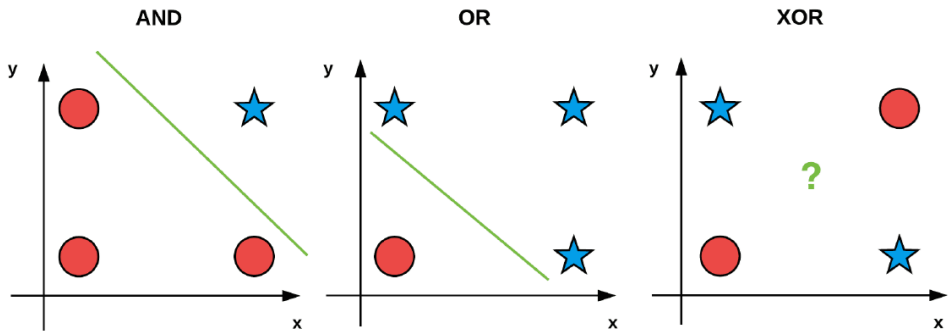
Contoh Kasus: Logika OR

Tabel kebenaran untuk fungsi OR adalah:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Implementasi Python

```
import numpy as np

# Data untuk logika OR
X = np.array([[0,0], [0,1], [1,0], [1,1]])
Y = np.array([0, 1, 1, 1])

# Inisialisasi bobot dan bias
w = np.zeros(2)
b = 0
eta = 0.1

def step(x):
    return 1 if x >= 0 else 0

# Training perceptron
for epoch in range(10):
    print(f"Epoch {epoch+1}")
    for x, y in zip(X, Y):
        z = np.dot(w, x) + b
        y_hat = step(z)
        error = y - y_hat
        w += eta * error * x
        b += eta * error
        print(f"Input: {x}, Target: {y}, Pred: {y_hat},
        Weights: {w}, Bias: {b}")
    print("-" * 40)

# Uji akhir
print("Hasil akhir prediksi:")
for x in X:
    z = np.dot(w, x) + b
    print(f"{x} => {step(z)}")
```



Gambar 1.2. Masalah XOR

1.3.1 Keterbatasan Perceptron: Masalah Linear Separability

Perceptron hanya dapat menyelesaikan masalah yang **linear separable**, artinya data dapat dipisahkan dengan satu garis lurus (atau bidang datar di dimensi lebih tinggi). Contohnya:

- Fungsi AND dan OR bisa dipisahkan secara linear.
- Fungsi XOR **tidak bisa** diselesaikan oleh perceptron tunggal karena tidak linear separable (Gambar 1.2).

Untuk menyelesaikan masalah seperti XOR, diperlukan lebih dari satu layer neuron — inilah yang melahirkan konsep **Multilayer Perceptron (MLP)** yang akan kita pelajari di bagian selanjutnya.

BAB 2

Fungsi Aktivasi dan Non-Linearitas

Pendahuluan

Dalam jaringan saraf tiruan, setiap neuron menghitung kombinasi linear dari input yang diterimanya, yaitu $\sum w_i x_i + b$. Namun jika hanya proses linear ini yang dilakukan, maka meskipun jaringan memiliki banyak lapisan, secara keseluruhan sistem tetap merupakan fungsi linear dan tidak mampu menangkap relasi yang kompleks dan non-linear dalam data.

Untuk mengatasi hal ini, jaringan saraf menggunakan **fungsi aktivasi (activation function)** setelah perhitungan linear, sehingga neuron dapat belajar fungsi yang bersifat non-linear. Inilah yang membuat jaringan saraf jauh lebih kuat dibanding model linear biasa.

2.1 Fungsi Aktivasi Populer

2.1.1 Fungsi Step

Fungsi ini digunakan pada perceptron klasik. Output-nya 0 atau 1, bergantung pada apakah input melebihi suatu ambang tertentu.

$$f(x) = \begin{cases} 1, & \text{jika } x \geq 0 \\ 0, & \text{jika } x < 0 \end{cases}$$

Namun, fungsi ini **tidak dapat digunakan** dalam jaringan modern karena tidak terdiferensiasi di titik 0 dan turunan di tempat lain bernilai nol.

2.1.2 Fungsi Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Turunannya:

$$\sigma'(x) = \sigma(x)\,(1 - \sigma(x))$$

Catatan: Turunan sigmoid mendekati nol jika x sangat besar atau sangat kecil, menyebabkan masalah *vanishing gradient*.

Fungsi sigmoid memetakan input real ke dalam rentang (0, 1), sehingga sering digunakan ketika output diharapkan sebagai probabilitas. Namun, fungsi ini mengalami

masalah **vanishing gradient**, terutama saat input bernilai sangat besar atau sangat kecil.

2.1.3 Fungsi Tanh

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Turunannya:

$$\tanh'(x) = 1 - \tanh^2(x)$$

Fungsi ini serupa dengan sigmoid, tetapi memetakan input ke rentang $(-1, 1)$. Biasanya memberikan performa yang lebih baik dibanding sigmoid karena data lebih terpusat di sekitar nol.

2.1.4 Fungsi ReLU (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

Turunannya:

$$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

ReLU adalah fungsi aktivasi paling populer di jaringan dalam (deep network) modern karena:

- Mudah dihitung
- Tidak mengalami vanishing gradient (kecuali untuk bagian negatif)
- Mempercepat konvergensi pelatihan

Namun, ReLU juga memiliki kekurangan, yaitu ketika banyak neuron berada di zona negatif, maka gradiennya menjadi nol dan menyebabkan **dead neuron**.

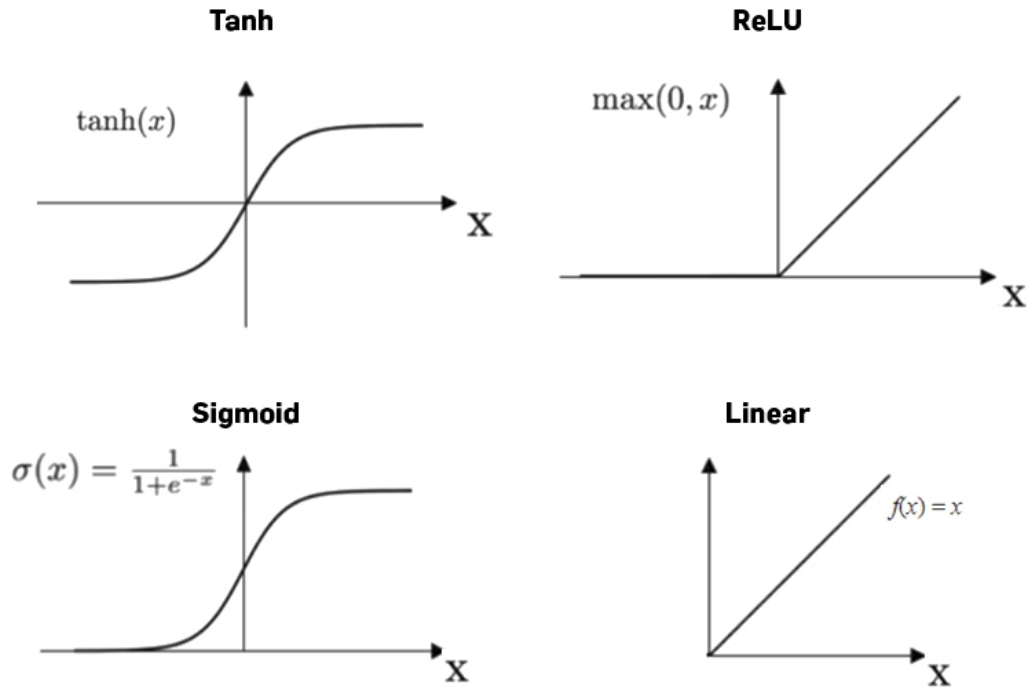
2.2 Perbandingan Fungsi Aktivasi

Fungsi	Rentang Output	Keunggulan	Kekurangan
Step	{0, 1}	Sederhana	Tidak terdiferensiasi
Sigmoid	(0, 1)	Interpretasi probabilitas	Vanishing gradient
Tanh	(-1, 1)	Lebih stabil dari sigmoid	Masih bisa vanishing gradient
ReLU	$[0, \infty)$	Cepat dan efisien	Bisa *dead neuron* (output selalu 0)

Gambar 2.1 menunjukkan beberapa jenis fungsi aktivasi.

2.3 Visualisasi Fungsi Aktivasi (Opsional)

Jika ingin membuat grafik fungsi-fungsi ini dalam Python:



Gambar 2.1. Plot beberapa fungsi aktivasi

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 100)
sigmoid = 1 / (1 + np.exp(-x))
tanh = np.tanh(x)
relu = np.maximum(0, x)

plt.plot(x, sigmoid, label='Sigmoid')
plt.plot(x, tanh, label='Tanh')
plt.plot(x, relu, label='ReLU')
plt.title("Fungsi Aktivasi")
plt.legend()
plt.grid()
plt.show()
```

—

BAB 3

Multilayer Perceptron dan Backpropagation

3.1 Pendahuluan: Konsep Multilayer Perceptron (MLP)

Setelah mempelajari perceptron tunggal, kita menyadari bahwa ada batasan besar: ia hanya bisa menyelesaikan masalah yang *linear separable*. Untuk menangani masalah yang lebih kompleks, kita memerlukan jaringan dengan lebih dari satu lapisan. Inilah yang disebut **Multilayer Perceptron (MLP)**.

- MLP terdiri dari **input layer**, satu atau lebih **hidden layer**, dan **output layer**.
- Setiap neuron dihubungkan penuh (*fully connected*) ke neuron di layer berikutnya.
- MLP menggunakan fungsi aktivasi non-linear seperti sigmoid atau ReLU pada hidden layer.

MLP memungkinkan pembelajaran dari relasi yang kompleks dan non-linear — misalnya fungsi XOR yang tidak dapat dipelajari oleh perceptron tunggal. Gambar 3.1 menunjukkan skema dasar arsitektur MLP.

Seluruh proses prediksi dilakukan secara *feedforward*, yaitu propagasi data dari input menuju output melalui layer-layer tersembunyi. Namun, untuk melatih MLP, kita memerlukan algoritma pelatihan khusus yang disebut **backpropagation**, yang akan dibahas di bab selanjutnya.

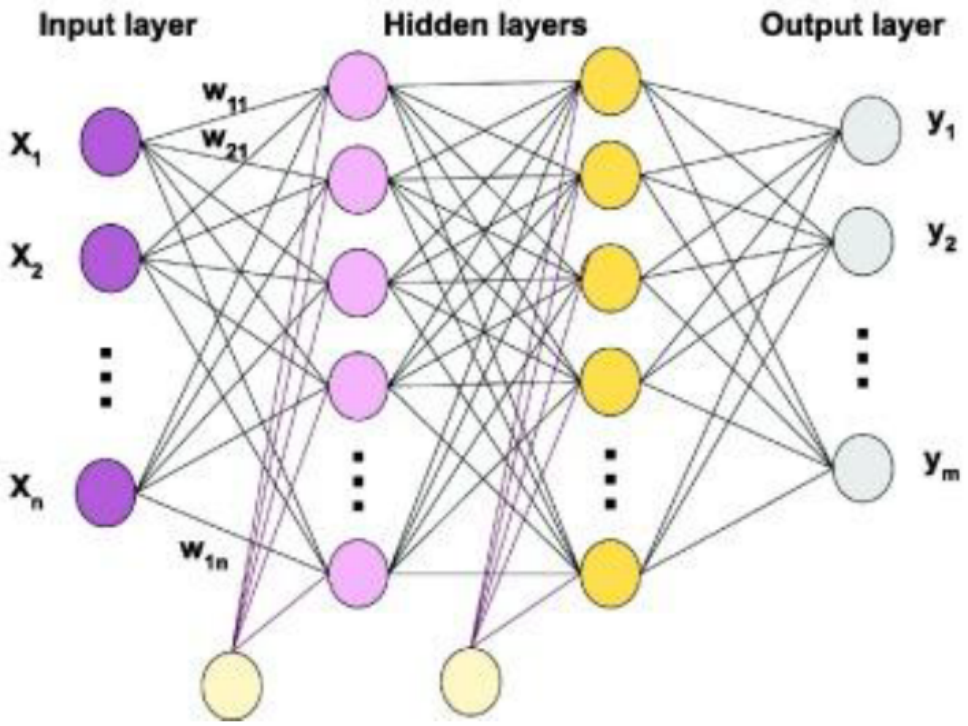
3.2 Arsitektur dan Proses Feedforward

Untuk memahami cara kerja MLP, kita akan mempelajari contoh sederhana jaringan saraf dengan:

- 2 neuron input
- 1 hidden layer dengan 3 neuron (aktivasi sigmoid)
- 1 neuron output (aktivasi sigmoid)

Kita akan menggunakan notasi sebagai berikut:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}, \quad a^{(l)} = \sigma\left(z^{(l)}\right)$$



Gambar 3.1. Struktur umum MLP

3.3 Struktur dan Alur MLP

Setiap neuron dalam suatu layer terhubung ke seluruh neuron di layer berikutnya (fully connected). Proses propagasi data dari input ke output disebut **feedforward**.

Sebagai contoh, kita gunakan arsitektur MLP dengan struktur:

- **2 input neuron** (untuk fitur x_1, x_2)
- **1 hidden layer** dengan 3 neuron (pakai aktivasi sigmoid)
- **1 output neuron** (sigmoid juga, untuk klasifikasi biner)

Langkah-langkah Feedforward:

1. Hitung output dari hidden layer:

$$z^{(1)} = XW^{(1)} + b^{(1)}, \quad a^{(1)} = \sigma\left(z^{(1)}\right)$$

2. Hitung output dari layer akhir:

$$z^{(2)} = a^{(1)}W^{(2)} + b^{(2)}, \quad a^{(2)} = \sigma\left(z^{(2)}\right)$$

dengan notasi:

- X: input berukuran $(n, 2)$
- $W^{(1)}$: bobot input $\rightarrow$ hidden, ukuran $(2, 3)$
- $b^{(1)}$: bias hidden, ukuran $(1, 3)$
- $W^{(2)}$: bobot hidden $\rightarrow$ output, ukuran $(3, 1)$
- $b^{(2)}$: bias output, ukuran $(1, 1)$
- $\sigma(\cdot)$: fungsi sigmoid

Proses ini disebut **feedforward propagation**.
Contoh perhitungan manual (arsitektur 2–3–1):
Misalkan:

- Input: $\vec{x} = \begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix}$
- Bobot dan bias untuk hidden layer (3 neuron):

$$W^{(1)} = \begin{bmatrix} 0.2 & -0.1 & 0.4 \\ 0.5 & 0.3 & -0.2 \end{bmatrix}, \quad b^{(1)} = \begin{bmatrix} 0.1 & 0.0 & -0.1 \end{bmatrix}$$

- Bobot dan bias untuk output layer (1 neuron):

$$W^{(2)} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.8 \end{bmatrix}, \quad b^{(2)} = 0.05$$

Langkah-langkah:

1. Hitung input ke hidden layer:

$$z^{(1)} = \vec{x}^{\top} W^{(1)} + b^{(1)} = \begin{bmatrix} 0.5 & 0.8 \end{bmatrix} \begin{bmatrix} 0.2 & -0.1 & 0.4 \\ 0.5 & 0.3 & -0.2 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.0 & -0.1 \end{bmatrix}$$

$$z^{(1)} = \begin{bmatrix} 0.5\,(0.2) + 0.8\,(0.5) + 0.1, & 0.5\,(-0.1) + 0.8\,(0.3) + 0.0, & 0.5\,(0.4) + 0.8\,(-0.2) - 0.1 \end{bmatrix}$$

$$z^{(1)} = \begin{bmatrix} 0.1 + 0.4 + 0.1, & -0.05 + 0.24 + 0, & 0.2 - 0.16 - 0.1 \end{bmatrix} = \begin{bmatrix} 0.6, & 0.19, & -0.06 \end{bmatrix}$$

2. Terapkan fungsi aktivasi sigmoid:

$$a^{(1)} = \sigma\left(z^{(1)}\right) = \begin{bmatrix} \sigma(0.6), & \sigma(0.19), & \sigma(-0.06) \end{bmatrix} \approx \begin{bmatrix} 0.645, & 0.547, & 0.485 \end{bmatrix}$$

3. Hitung output:

$$z^{(2)} = a^{(1)} \cdot W^{(2)} + b^{(2)} = \begin{bmatrix} 0.645 & 0.547 & 0.485 \end{bmatrix} \begin{bmatrix} 0.6 \\ -0.4 \\ 0.8 \end{bmatrix} + 0.05$$

$$z^{(2)} = (0.645)\,(0.6) + (0.547)\,(-0.4) + (0.485)\,(0.8) + 0.05 \approx 0.387 - 0.219 + 0.388 + 0.05 = 0.606$$

4. Output akhir (setelah sigmoid):

$$a^{(2)} = \sigma(0.606) \approx 0.647$$

Implementasi Python: Proses feedforward di atas dapat diimplementasikan dalam kode Python sederhana sebagai berikut:

```
import numpy as np

# Input vektor (2x1)
x = np.array([[0.5], [0.8]])

# Hidden layer: 3 neuron
W1 = np.array([[0.2, 0.4, -0.3],
               [-0.5, 0.3, 0.6]])   # (2x3)
b1 = np.array([[0.1], [0.2], [0.05]])  # (3x1)
z1 = W1.T @ x + b1      # (3x1)
a1 = 1 / (1 + np.exp(-z1))  # sigmoid aktivasi (3x1)

# Output layer: 1 neuron
W2 = np.array([[0.7, -1.2, 0.5]])  # (1x3)
b2 = np.array([[0.05]])           # (1x1)
z2 = W2 @ a1 + b2                 # (1x1)
a2 = 1 / (1 + np.exp(-z2))        # sigmoid akhir

print("Output prediksi:", a2)
```

3.4 Backpropagation dan Gradient Descent

Setelah kita memahami alur *feedforward*, sekarang kita bahas bagaimana jaringan saraf belajar dari data. Proses pembelajaran dilakukan dengan menyesuaikan bobot dan bias untuk meminimalkan kesalahan prediksi. Proses ini disebut **training**, dan algoritma utama yang digunakan adalah **backpropagation**.

Apa itu Backpropagation?

Backpropagation (propagasi mundur) adalah algoritma untuk menghitung **turunan** (gradien) dari fungsi kerugian terhadap bobot-bobot dalam jaringan saraf. Ini dilakukan dengan menggunakan aturan rantai (*chain rule*) dari kalkulus.

Tujuan dari backpropagation adalah mengetahui:

$$\frac{\partial L}{\partial w_{ij}^{(l)}}$$

yaitu perubahan loss L terhadap bobot $w_{ij}^{(l)}$ di layer ke-l, antara neuron ke-i dan ke-j.

Hubungan dengan Gradient Descent

Backpropagation sendiri hanya menghitung gradien. Untuk benar-benar memperbarui bobot, kita gunakan metode **gradient descent**:

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

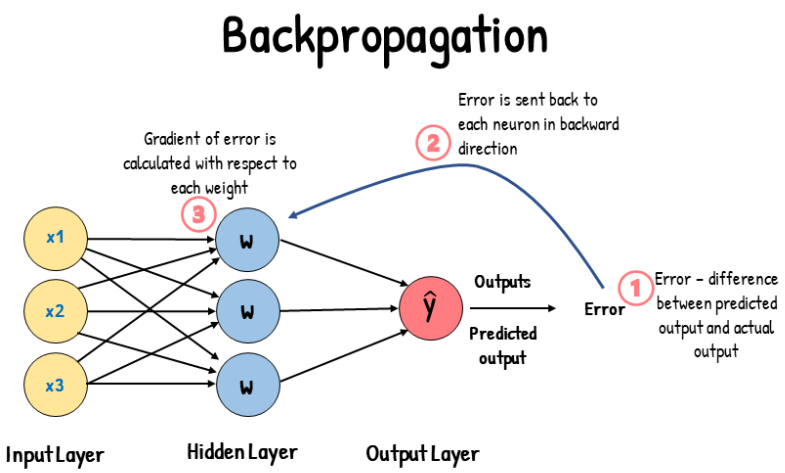
$$b \leftarrow b - \eta \cdot \frac{\partial L}{\partial b}$$

dengan η adalah **learning rate**, yaitu seberapa besar perubahan yang kita lakukan pada setiap langkah.

Langkah-langkah Backpropagation:

1. **Forward pass**: hitung output jaringan 2. **Hitung error**: selisih antara output dan label 3. **Backward pass**: gunakan chain rule untuk menghitung turunan 4. **Update parameter**: gunakan gradient descent

Ilustrasi alur ini digambarkan dalam Gambar 3.2.



Gambar 3.2. Alur umum proses feedforward dan backpropagation (sumber: analyticsvidhya.com)

Fungsi Kerugian (Loss Function)

Untuk menghitung seberapa buruk prediksi, digunakan fungsi kerugian.

- Untuk regresi: **Mean Squared Error (MSE)**

$$L = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- Untuk klasifikasi (output sigmoid): **Binary Cross-Entropy**

$$L = -\left(y \log\left(\hat{y}\right) + (1 - y) \log\left(1 - \hat{y}\right)\right)$$

Propagasi Turunan di Jaringan

Misalkan $a^{(l)}$ adalah aktivasi pada layer l, dan $z^{(l)}$ adalah nilai inputnya:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}, \quad a^{(l)} = f\left(z^{(l)}\right)$$

Dengan rantai turunan:

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)}\left(a^{(l-1)}\right)^T$$

$$\delta^{(l)} = \left(W^{(l+1)T}\delta^{(l+1)}\right) \circ f'\left(z^{(l)}\right)$$

Untuk layer output:

$$\delta^{(L)} = \frac{\partial L}{\partial a^{(L)}} \circ f'\left(z^{(L)}\right)$$

Di mana: - $\delta^{(l)}$ disebut **error term** untuk layer ke-l - $\circ$ adalah operasi Hadamard (perkalian elemen per elemen)

3.5 Implementasi MLP untuk XOR

Berikut adalah implementasi sederhana jaringan 2-3-1 untuk mempelajari fungsi XOR. Kode ini menghitung forward dan backward pass secara manual tanpa library eksternal. Jika dijalankan, maka seharusnya kita akan mendapatkan akurasi sempurna (100%) untuk permasalahan XOR.

```
import numpy as np

# Fungsi aktivasi sigmoid dan turunannya
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_deriv(x):
    s = sigmoid(x)
    return s * (1 - s)

# Data latih XOR
X = np.array([[0,0],[0,1],[1,0],[1,1]])
Y = np.array([[0],[1],[1],[0]])

# Inisialisasi bobot dan bias
np.random.seed(42)
w1 = np.random.randn(2, 3)   # 2 input → 3 hidden
b1 = np.zeros((1, 3))
w2 = np.random.randn(3, 1)   # 3 hidden → 1 output
b2 = np.zeros((1, 1))

# Proses pelatihan dengan backpropagation
lr = 0.1
```

```
for epoch in range(10000):
    # Feedforward
    z1 = X @ w1 + b1
    a1 = sigmoid(z1)
    z2 = a1 @ w2 + b2
    a2 = sigmoid(z2)

    # Backpropagation
    error = a2 - Y
    d_z2 = error * sigmoid_deriv(z2)
    d_w2 = a1.T @ d_z2
    d_b2 = np.sum(d_z2, axis=0, keepdims=True)

    d_a1 = d_z2 @ w2.T
    d_z1 = d_a1 * sigmoid_deriv(z1)
    d_w1 = X.T @ d_z1
    d_b1 = np.sum(d_z1, axis=0, keepdims=True)

    # Update bobot dan bias
    w1 -= lr * d_w1
    b1 -= lr * d_b1
    w2 -= lr * d_w2
    b2 -= lr * d_b2

    # Cetak loss tiap 2000 epoch
    if epoch % 2000 == 0:
        loss = np.mean((a2 - Y)**2)
        print(f"Epoch {epoch}, Loss: {loss:.4f}")


# Evaluasi akhir
z1 = X @ w1 + b1
a1 = sigmoid(z1)
z2 = a1 @ w2 + b2
a2 = sigmoid(z2)

print("\nPrediksi akhir (dibulatkan):")
print(a2.round().astype(int).ravel())

print("Label asli:")
print(Y.ravel())

akurasi = np.mean((a2.round().astype(int).ravel()) == Y.ravel())
print(f"Akurasi akhir: {akurasi:.2f}")
```

3.6 Implementasi MLP dengan `scikit-learn`

Selain membangun MLP dari awal secara manual, kita juga bisa menggunakan pustaka Python seperti `scikit-learn` untuk membangun dan melatih model MLP dengan lebih mudah dan efisien.

Dalam contoh ini, kita akan menggunakan **dataset Iris**, yang merupakan dataset klasik untuk klasifikasi tiga kelas berdasarkan panjang dan lebar sepal dan petal bunga Iris.

Contoh: Klasifikasi Dataset Iris dengan MLP

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Bagi data menjadi data latih dan uji
X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.3, random_state=42)

# Normalisasi fitur
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Bangun dan latih model MLP
mlp = MLPClassifier(hidden_layer_sizes=(10,), activation='relu',
                    solver='adam', max_iter=1000, random_state=42)

mlp.fit(X_train, y_train)

# Evaluasi model
y_pred = mlp.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Penjelasan:

- `hidden_layer_sizes=(10,)` artinya ada 1 hidden layer dengan 10 neuron.
- `activation='relu'` menggunakan fungsi aktivasi ReLU.
- `solver='adam'` adalah algoritma optimasi yang digunakan untuk backpropagation.
- Data diproses dengan `StandardScaler` agar fitur-fitur memiliki skala yang seimbang.

Output: Program ini akan menampilkan matriks kebingungan dan metrik evaluasi seperti akurasi, precision, recall, dan F1-score untuk masing-masing kelas pada data Iris.

Catatan

Walaupun implementasi ini sangat singkat dan efisien, penting untuk tetap memahami prinsip dasar feedforward, aktivasi, dan backpropagation seperti yang telah dibahas sebelumnya. Dengan pemahaman tersebut, kita dapat lebih bijak memilih arsitektur, fungsi aktivasi, serta algoritme pelatihan yang tepat.

Penutup

MLP membuka jalan bagi jaringan saraf dalam yang lebih kompleks. Namun, untuk bisa benar-benar melatih jaringan besar, kita butuh strategi yang efisien dan stabil — seperti fungsi aktivasi yang tepat, normalisasi, optimisasi yang lebih baik (misalnya Adam), dan arsitektur lanjutan yang akan dibahas di bab-bab berikutnya.

BAB 4

Perkembangan Deep Learning Modern

Pendahuluan

Setelah mempelajari perceptron dan multilayer perceptron (MLP), kita melihat bagaimana jaringan saraf dapat digunakan untuk mempelajari pola dari data. Namun, seiring meningkatnya kompleksitas data seperti gambar, teks, dan sinyal suara, pendekatan klasik MLP menjadi kurang memadai. Di sinilah **Deep Learning** mengambil peran.

Deep learning adalah pendekatan modern yang menggunakan jaringan saraf dengan banyak lapisan (*deep neural networks*) untuk mengekstrak representasi hierarkis dari data.

4.1 Keterbatasan MLP Klasik

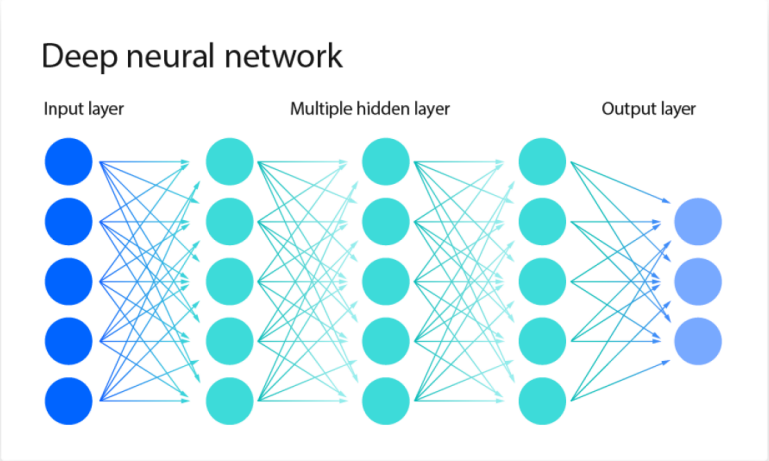
MLP klasik memiliki beberapa keterbatasan:

- **Jumlah parameter terlalu besar** jika digunakan pada input berdimensi tinggi (misalnya gambar 100x100 pixel akan memiliki 10.000 input neuron).
- **Tidak memanfaatkan struktur spasial atau sekuensial data**, sehingga tidak efisien untuk data seperti gambar atau teks.
- **Sulit dilatih jika terlalu dalam**, karena masalah seperti *vanishing gradient*.

4.2 Deep Learning sebagai Solusi

Deep learning modern mengatasi keterbatasan tersebut dengan:

- Menggunakan arsitektur khusus yang disesuaikan dengan jenis data: Convolutional Neural Network (CNN) untuk data gambar/citra, Recurrent Neural Network (RNN) serta Long Short Term Memory (LSTM) untuk data sekuensial ataupun deret waktu, dan Transformer untuk data urutan dengan konteks global (misalnya bahasa alami).
- Memanfaatkan teknik normalisasi dan regularisasi untuk meningkatkan stabilitas pelatihan.



Gambar 4.1. Deep Learning

- Menggunakan fungsi aktivasi non-linear yang lebih baik (seperti ReLU dan varian-nya).
- Didukung oleh perangkat keras (GPU/TPU) dan pustaka perangkat lunak (seperti TensorFlow dan PyTorch) yang sangat efisien.

4.3 Ciri-ciri Jaringan Deep Learning

Berikut ini beberapa karakteristik umum dari deep learning modern:

- **Banyak Lapisan**: terdiri dari puluhan hingga ratusan lapisan.
- **Representasi Hierarkis**: lapisan awal belajar fitur sederhana, lapisan atas belajar konsep kompleks.
- **Transfer Learning**: model yang telah dilatih pada dataset besar dapat digunakan kembali pada tugas yang berbeda.
- **End-to-End Learning**: tidak perlu eksplisit melakukan feature engineering — model belajar langsung dari data mentah.

4.4 Jenis Data dalam Deep Learning

Deep learning modern mampu menangani berbagai jenis data:

- **Citra dan Video**: menggunakan CNN dan arsitektur konvolusional.
- **Data Urutan/*Time Series* (Teks, Audio)**: menggunakan RNN, LSTM, GRU, dan Transformer.

- **Data Graf dan Spasial**: menggunakan Graph Neural Networks (GNN).
- **Multimodal**: menggabungkan beberapa jenis data sekaligus (misalnya gambar + teks).

4.5 Perbandingan Arsitektur Deep Learning

Aspek	**MLP**	**CNN**	**RNN / LSTM**
Input Umum	Data tabular	Citra/video	Teks/waktu/audio
Struktur Input	Flat (vektor)	2D (grid)	Urutan (sequence)
Sifat Fitur	Tidak berstruktur	Spasial (lokalitas)	Temporal (urutan)
Reusabilitas Parameter	Tidak ada	Ada (shared filter)	Ada (recurrent weights)
Ukuran Model	Besar	Cocok untuk gambar	Cocok untuk timeseries

4.6 Masalah Vanishing Gradient dan Solusinya

Dalam jaringan yang sangat dalam, turunan (gradien) dari fungsi aktivasi seringkali menjadi sangat kecil saat dihitung mundur (backpropagation), terutama jika menggunakan fungsi seperti sigmoid. Ini disebut **vanishing gradient**, dan membuat pembelajaran sangat lambat atau bahkan berhenti sama sekali.

Solusi umum untuk masalah ini:

- Menggunakan fungsi aktivasi seperti ReLU (Rectified Linear Unit), yang memiliki turunan konstan untuk nilai positif.
- Menggunakan teknik **Batch Normalization** untuk menstabilkan distribusi aktivasi selama pelatihan.
- Menggunakan optimizers modern seperti Adam.

4.7 Transfer Learning dan Pretraining

Transfer learning adalah strategi pelatihan di mana model yang telah dilatih pada satu tugas (misalnya mengenali objek di dataset ImageNet) digunakan kembali untuk tugas lain (misalnya klasifikasi daun tanaman).

- Biasanya dilakukan dengan menggunakan model *pra-latih* dan menambahkan lapisan baru di atasnya.
- Sangat efisien karena tidak perlu melatih seluruh model dari awal.

Contoh: Model CNN pretrained seperti ResNet atau MobileNet dapat digunakan untuk klasifikasi objek, bahkan dengan dataset kecil.

4.8 Aplikasi Deep Learning Modern

Deep learning telah digunakan dalam berbagai aplikasi dunia nyata:

- **Kendaraan otonom (self-driving cars)**
- **Pendeteksi penyakit dari citra medis** (misalnya kanker kulit dari foto dermatoskopi)
- **Pengenal suara dan asisten virtual** (Google Assistant, Siri)
- **Terjemahan mesin otomatis** (Google Translate berbasis Transformer)
- **Deteksi penipuan (fraud detection)** dari pola transaksi keuangan

4.9 Evolusi Perangkat Keras dan Perangkat Lunak

Kemajuan deep learning sangat didukung oleh perkembangan teknologi:

- **GPU dan TPU**: memungkinkan pelatihan jaringan besar dengan lebih cepat
- **Library populer**: `TensorFlow`, `Keras`, `PyTorch` membuat pengembangan model jauh lebih mudah
- **Ekosistem**: Jupyter Notebook, Google Colab, Hugging Face Transformers, dll.

4.10 Representasi Bertingkat

Salah satu kekuatan deep learning adalah kemampuannya membangun **representasi bertingkat**. Dalam CNN, misalnya:

- Lapisan pertama mendeteksi *edge*
- Lapisan berikutnya mengenali *bentuk*
- Lapisan atas mengidentifikasi *objek kompleks*

Ini meniru cara kerja otak manusia dalam memproses informasi visual — dari deteksi bentuk dasar ke objek utuh.

Penutup

Deep learning bukan sekadar jaringan dengan banyak lapisan, tetapi merupakan pendekatan sistematis yang mengandalkan struktur arsitektur dan metode pelatihan yang dirancang khusus untuk menangani tipe data tertentu. Dalam bab-bab berikutnya, kita akan membahas berbagai arsitektur jaringan deep learning seperti CNN, RNN, dan Transformer, serta aplikasinya dalam computer vision dan natural language processing.

BAB 5

Arsitektur Dasar Deep Learning

Pendahuluan

Seiring berkembangnya deep learning, para peneliti AI mengembangkan berbagai jenis arsitektur jaringan saraf yang disesuaikan dengan karakteristik data yang berbeda. Beberapa arsitektur paling berpengaruh dan digunakan luas hingga kini meliputi:

- **Convolutional Neural Networks (CNN)**: untuk pengolahan gambar dan video
- **Recurrent Neural Networks (RNN)**: untuk data berurutan seperti teks dan suara
- **Long Short-Term Memory (LSTM)** dan **GRU**: varian dari RNN untuk mengatasi kendala jangka panjang
- **Autoencoder**: arsitektur yang digunakan untuk representasi laten, reduksi dimensi dan pemodelan *unsupervised*
- **Transformer**: arsitektur yang saat ini mendominasi NLP dan multimodal AI

Dalam bab ini, kita akan mulai dengan membahas CNN, dan dilanjutkan dengan RNN, LSTM & GRU serta Autoencoder. Transformer akan dibahas pada modul berikutnya.

5.1 Convolutional Neural Networks (CNN)

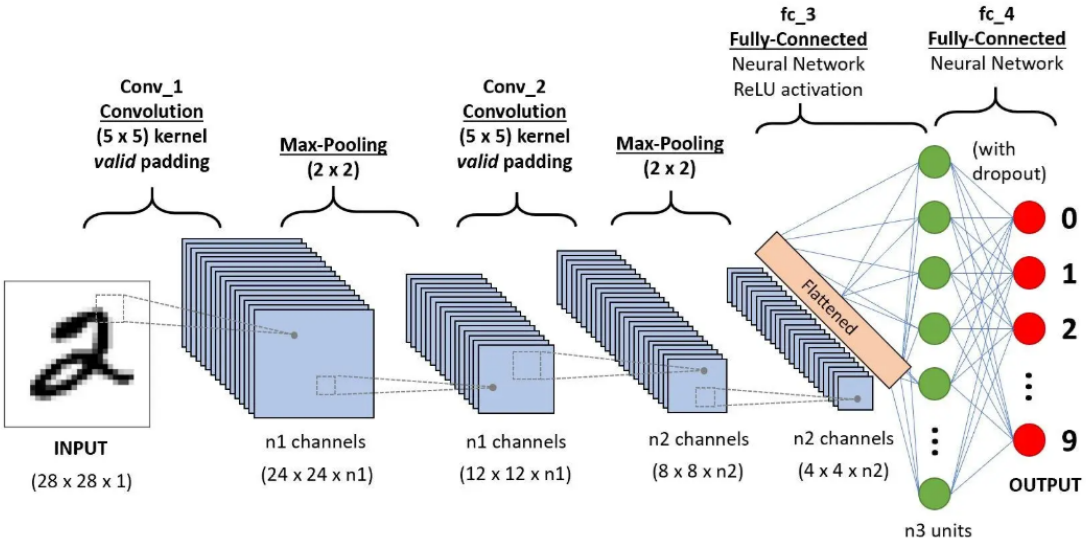
5.1.1 Motivasi dan Intuisi

Convolutional Neural Network (CNN) merupakan arsitektur jaringan saraf yang dirancang khusus untuk menangani data citra atau data lain yang memiliki struktur spasial. Pada MLP, semua neuron saling terhubung, sehingga jumlah parameter menjadi sangat besar dan tidak efisien untuk data berdimensi tinggi seperti gambar.

CNN mengatasi masalah ini dengan menggunakan konsep **filter (atau kernel)** yang berukuran kecil dan digeser (*sliding*) pada input. Filter ini mendeteksi pola lokal seperti tepi, sudut, atau tekstur. Dengan cara ini, CNN mampu mengekstraksi fitur penting secara otomatis dan efisien.

Contoh intuitif: Jika MLP "melihat seluruh gambar sekaligus", maka CNN seperti "melihat gambar menggunakan jendela kecil" lalu mencatat bagian penting dari gambar tersebut satu per satu.

CNN juga memiliki keunggulan berupa:



Gambar 5.1. Ilustrasi CNN untuk pengenalan tulisan angka (Sumber:https://saturncloud.io/)

- Jumlah parameter lebih sedikit (karena parameter dibagi antar posisi)
- Lebih tahan terhadap pergeseran atau rotasi kecil pada gambar
- Dapat menangkap informasi spasial secara bertingkat

Gambar 5.1 menunjukkan arsitektur sebuah CNN untuk pengenalan angka dari tulisan tangan.

5.1.2 Struktur CNN

Sebuah jaringan CNN terdiri dari beberapa lapisan utama yang bekerja bersama untuk mengekstraksi dan memproses informasi dari input citra:

1. **Lapisan Konvolusi (Convolutional Layer)**
   Melakukan operasi konvolusi antara input dan filter (kernel) berukuran kecil (misalnya 3x3 atau 5x5). Setiap filter bertugas mendeteksi pola tertentu, dan hasilnya disebut *feature map*.
2. **Fungsi Aktivasi**
   Biasanya digunakan fungsi ReLU (Rectified Linear Unit) untuk memperkenalkan non-linearitas, yang memungkinkan jaringan belajar pola kompleks.
3. **Lapisan Pooling (Pooling Layer)**
   Pooling mengecilkan dimensi spasial dari feature map, sehingga mengurangi jumlah parameter dan memperkuat ketahanan terhadap pergeseran. Contohnya adalah *max pooling* dan *average pooling*.
4. **Lapisan Fully Connected (Dense Layer)**
   Setelah beberapa lapisan konvolusi dan pooling, hasilnya diubah menjadi vektor

dan diteruskan ke lapisan dense seperti pada MLP untuk prediksi akhir.

Struktur umum CNN dapat divisualisasikan sebagai berikut:

$$\texttt{Input Image} \rightarrow \texttt{[Conv} \rightarrow \texttt{ReLU} \rightarrow \texttt{Pool]}^{\times n} \rightarrow \texttt{Flatten} \rightarrow \texttt{Dense} \rightarrow \texttt{Output}$$

Contoh jaringan CNN pada dataset MNIST (citra digit 28x28):

- Conv2D: 32 filter 3x3
- ReLU
- MaxPooling 2x2
- Conv2D: 64 filter 3x3
- ReLU
- MaxPooling 2x2
- Flatten
- Dense: 128 neuron
- Output layer: 10 neuron (softmax)

5.1.3 Operasi Konvolusi

Operasi konvolusi merupakan inti dari CNN. Ia mengambil input berupa matriks (biasanya representasi citra), lalu menggeser **filter (kernel)** kecil untuk menghasilkan fitur baru. Filter berukuran 3x3 atau 5x5 ini secara sistematis digeser (stride) ke seluruh bagian citra dan dikalikan elemen demi elemen dengan area lokal pada citra.

Secara matematis, untuk input citra dua dimensi I dan filter K ukuran $f \times f$, output dari operasi konvolusi di titik (i, j) adalah:

$$S\left(i,j\right) = \sum_{m=0}^{f-1} \sum_{n=0}^{f-1} I\left(i+m, j+n\right) \cdot K\left(m,n\right)$$

Misalnya, input citra 5x5 dan filter 3x3:

$$I = \begin{bmatrix} 1 & 2 & 0 & 3 & 1 \\ 4 & 1 & 0 & 1 & 7 \\ 1 & 2 & 3 & 2 & 1 \\ 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 1 & 0 & 0 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Langkah pertama (pada pojok kiri atas input):

$$\begin{aligned} S\left(0,0\right) &= (1 \cdot 1) + (2 \cdot 0) + (0 \cdot -1) \\ &\quad + (4 \cdot 1) + (1 \cdot 0) + (0 \cdot -1) \\ &\quad + (1 \cdot 1) + (2 \cdot 0) + (3 \cdot -1) \\ &= 1 + 0 + 0 + 4 + 0 + 0 + 1 + 0 - 3 \\ &= 3 \end{aligned}$$

Operasi ini diulang untuk setiap posisi filter di seluruh area input. Hasil akhirnya disebut **feature map**.

5.1.3.0.1 Catatan tentang Padding dan Stride

- **Padding**: menambahkan batas nol di tepi input agar ukuran output tetap sama dengan input.
- **Stride**: menentukan seberapa jauh filter digeser. Stride 1 = geser 1 piksel; Stride 2 = geser 2 piksel, dan seterusnya.

5.1.4 Pooling

Pooling adalah proses untuk mengecilkan dimensi spasial dari feature map yang dihasilkan oleh lapisan konvolusi. Tujuan utama pooling adalah untuk:

- Mengurangi jumlah parameter dan komputasi
- Membuat representasi fitur menjadi lebih stabil terhadap perubahan kecil seperti pergeseran dan rotasi
- Menghindari overfitting dengan menyederhanakan informasi

Jenis pooling yang paling umum adalah:

1. **Max Pooling**: memilih nilai maksimum dari sebuah jendela lokal, misalnya 2x2
2. **Average Pooling**: menghitung rata-rata dari elemen dalam jendela lokal

Contoh Max Pooling dengan jendela 2x2 dan stride 2 pada matriks 4x4:

$$\text{Input} = \begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 1 & 2 \\ 9 & 1 & 8 & 7 \\ 4 & 2 & 0 & 3 \end{bmatrix} \quad \text{Output (Max Pooling)} = \begin{bmatrix} 6 & 4 \\ 9 & 8 \end{bmatrix}$$

Pooling tidak memiliki parameter yang perlu dipelajari. Ia hanya menerapkan operasi statistik sederhana untuk memperkecil representasi spasial tanpa mengubah jumlah channel (kedalaman) data.

Lapisan pooling biasanya ditempatkan setelah satu atau dua lapisan konvolusi untuk mengontrol dimensi keluaran dari jaringan.

5.1.5 Contoh Arsitektur CNN dan Implementasi (dengan PyTorch)

Sebagai contoh implementasi CNN, kita akan membangun jaringan konvolusi sederhana menggunakan PyTorch untuk dataset MNIST.

Struktur CNN

Arsitektur model:

- Conv2D: 32 filter 3x3, activation ReLU
- MaxPool2D: 2x2
- Conv2D: 64 filter 3x3, activation ReLU
- MaxPool2D: 2x2
- Flatten
- Dense (Linear): 128 neuron, ReLU
- Output layer: 10 neuron (Softmax untuk klasifikasi digit 0–9)

Kode Implementasi (dengan PyTorch)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Transformasi data: normalisasi ke [0,1] dan ubah ke tensor
transform = transforms.Compose([
    transforms.ToTensor()
])

# Load dataset MNIST
train_data = datasets.MNIST(root='./data',
        train=True, download=True, transform=transform)
test_data = datasets.MNIST(root='./data',
        train=False, download=True, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64)

# Definisi CNN
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3)
            # input channel 1 (grayscale), output 32
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3)
        self.fc1 = nn.Linear(64 * 5 * 5, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))   # [batch, 32, 13, 13]
        x = self.pool(F.relu(self.conv2(x)))   # [batch, 64, 5, 5]
        x = x.view(-1, 64 * 5 * 5)              # Flatten
        x = F.relu(self.fc1(x))
```

```
        x = self.fc2(x)
        return x

# Inisialisasi model, loss, optimizer
model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training loop (sederhana)
for epoch in range(5):
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(train_loader):.4f}")
```

Model ini dapat diperluas lebih lanjut dengan menambahkan dropout, batch normalization, atau data augmentation jika dibutuhkan.

5.2 Recurrent Neural Networks

Recurrent Neural Networks (RNN) adalah arsitektur jaringan saraf yang dirancang khusus untuk menangani data yang bersifat sekuensial, seperti teks, suara, sinyal waktu, atau data deret waktu lainnya. Berbeda dengan jaringan feedforward biasa (seperti MLP atau CNN), RNN memiliki koneksi berulang yang memungkinkan informasi dari waktu sebelumnya memengaruhi keluaran saat ini (Gambar 5.2).

Dalam arsitektur RNN, keluaran dari satu langkah waktu akan menjadi masukan tambahan untuk langkah waktu berikutnya. Dengan demikian, RNN dapat memiliki memori terhadap elemen-elemen sebelumnya dalam urutan. Perhatikan Gambar 5.3 yang menunjukkan operasi yang terjadi ketika kita "membentangkan" (*unroll*) operasi yang dilakukan RNN dalam sumbu waktu.

Definisi Model RNN

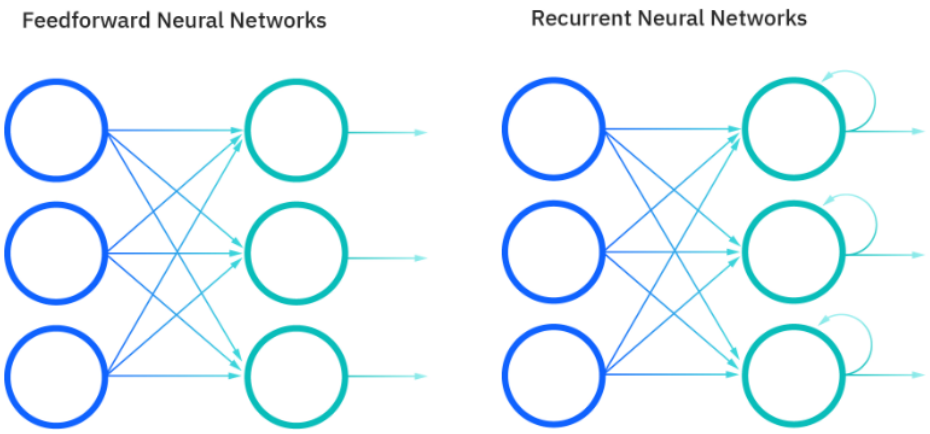
Secara matematis, RNN memproses input sekuensial $x^{(t)}$ pada waktu ke-t dengan rumus:

$$h^{(t)} = \sigma\left(W_{hx}x^{(t)} + W_{hh}h^{(t-1)} + b_h\right)$$

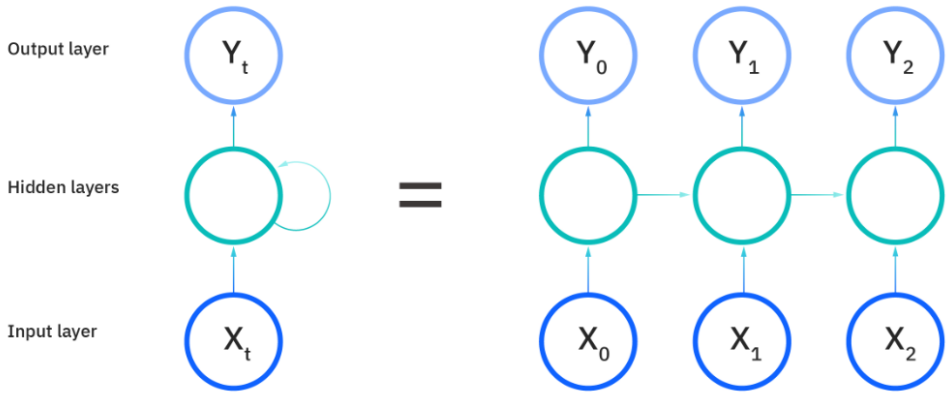
$$y^{(t)} = W_{hy}h^{(t)} + b_y$$

Dengan:

- $x^{(t)}$ = input pada waktu ke-t



Gambar 5.2. Feedforward NN vs RNN (sumber:ibm.com)



Gambar 5.3. *Unroll* operasi RNN (sumber: ibm.com)

- $h^{(t)}$ = hidden state pada waktu ke-t
- $y^{(t)}$ = output pada waktu ke-t
- W_{hx}, W_{hh}, W_{hy} = matriks bobot
- b_h, b_y = bias
- σ = fungsi aktivasi, biasanya tanh atau ReLU

Masalah pada RNN Sederhana

Meskipun RNN dapat menangani sekuens pendek, model ini memiliki keterbatasan dalam mengingat dependensi jangka panjang. Hal ini disebabkan oleh masalah **vanishing gradient** saat proses backpropagation melalui waktu (*Backpropagation Through Time / BPTT*). Gradien yang sangat kecil akan membuat bobot tidak ter-update dengan baik, menyebabkan hilangnya memori jangka panjang.

Masalah ini mendorong pengembangan varian RNN yang lebih kompleks seperti LSTM dan GRU yang akan dibahas di bab berikutnya.

5.3 Contoh Implementasi RNN Sederhana

Sebagai ilustrasi sederhana, kita akan menggunakan RNN untuk memprediksi kelanjutan dari deret sinus. Ini adalah contoh klasik yang menunjukkan bagaimana RNN mempelajari pola berurutan.

Persiapan Data: Deret Sinus

```
import numpy as np
import torch
from torch import nn
import matplotlib.pyplot as plt

# Bangkitkan data sinus
x = np.linspace(0, 100, 1000)
y = np.sin(x)

# Normalisasi ke bentuk [batch, time_step, feature]
def create_sequences(data, seq_length):
    sequences = []
    targets = []
    for i in range(len(data) - seq_length):
        sequences.append(data[i:i+seq_length])
        targets.append(data[i+seq_length])
    return np.array(sequences), np.array(targets)

SEQ_LEN = 50
X, Y = create_sequences(y, SEQ_LEN)
X = X[..., np.newaxis]  # tambahkan dimensi fitur
```

```
# Convert to tensor
X_tensor = torch.tensor(X, dtype=torch.float32)
Y_tensor = torch.tensor(Y, dtype=torch.float32).unsqueeze(1)
```

Model RNN dengan PyTorch

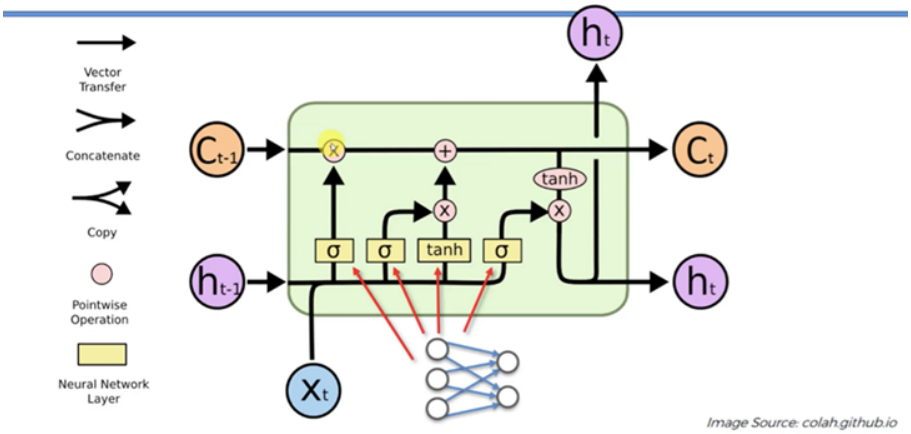
```
class SimpleRNN(nn.Module):
    def __init__(self, input_size=1, hidden_size=20, output_size=1):
        super(SimpleRNN, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :])  # Ambil hanya output terakhir
        return out

model = SimpleRNN()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

Training Model

```
EPOCHS = 30
for epoch in range(EPOCHS):
    output = model(X_tensor)
    loss = criterion(output, Y_tensor)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 5 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

Plot Hasil Prediksi

```
# Prediksi
predicted = model(X_tensor).detach().numpy()

plt.figure(figsize=(10,4))
plt.plot(Y, label='Asli')
plt.plot(predicted, label='Prediksi')
plt.legend()
plt.title("Prediksi Deret Sinus dengan RNN")
plt.show()
```



Gambar 5.4. Ilustrasi struktur LSTM (sumber: colah.github.io)

Model ini secara kasar mempelajari pola sinusoidal dan menunjukkan bagaimana RNN dapat digunakan untuk memahami struktur sekuens.

Selanjutnya, kita akan membahas varian yang lebih canggih dari RNN, yaitu **LSTM** dan **GRU**, yang dapat mengatasi kelemahan RNN klasik dalam menangani dependensi jangka panjang.

5.4 LSTM

Motivasi: Mengatasi Vanishing Gradient pada RNN

Seperti telah dijelaskan sebelumnya, RNN klasik memiliki kelemahan dalam mempertahankan informasi jangka panjang karena terjadinya *vanishing gradient* selama pelatihan menggunakan *Backpropagation Through Time (BPTT)*. Hal ini membuat RNN sulit mempelajari ketergantungan jangka panjang antar elemen sekuens.

Untuk mengatasi hal ini, Hochreiter dan Schmidhuber (1997) memperkenalkan **Long Short-Term Memory (LSTM)** yang memperkenalkan *gating mechanism* untuk mengatur aliran informasi.

Arsitektur Internal LSTM

Gambar 5.4 adalah ilustrasi struktur LSTM.

LSTM memiliki 4 komponen utama:

- **Forget Gate** (f_t): Memutuskan informasi mana dari cell state sebelumnya yang akan dibuang.
- **Input Gate** (i_t): Memutuskan informasi baru apa yang akan ditambahkan ke cell state.

- **Candidate Memory ($\tilde{C}_t$)**: Representasi baru dari informasi yang akan ditambahkan.
- **Output Gate (o_t)**: Memutuskan bagian mana dari cell state yang akan dikeluarkan sebagai output (hidden state).

Persamaan Perhitungan LSTM

$$
\begin{aligned}
f_t &= \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\
\tilde{C}_t &= \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \\
h_t &= o_t \odot \tanh\left(C_t\right)
\end{aligned}
$$

Dengan:

- x_t : input pada waktu ke-t
- h_{t-1} : hidden state sebelumnya
- C_{t-1} : cell state sebelumnya
- C_t : cell state saat ini
- h_t : hidden state saat ini (juga output)
- W_* dan b_* : bobot dan bias untuk masing-masing gate
- σ : sigmoid, $\odot$: perkalian elemenwise

Penjelasan Aliran Informasi

1. **Forget Gate (f_t)** menentukan berapa banyak informasi dari memori sebelumnya (C_{t-1}) yang dilupakan.
2. **Input Gate dan Candidate (i_t dan $\tilde{C}_t$)** menentukan berapa banyak informasi baru ($\tilde{C}_t$) yang ditambahkan ke memori saat ini (C_t).
3. **Output Gate (o_t)** menentukan bagian dari memori C_t yang akan diproyeksikan menjadi output h_t.

Struktur ini memungkinkan gradien mengalir secara lebih stabil dalam waktu, sehingga LSTM mampu mengingat informasi dalam urutan panjang.

Ilustrasi Naratif

Misalnya kita memproses kalimat:

> "Dia memakan apel karena **dia** lapar."

Ketika kita sampai pada kata "dia" yang kedua, LSTM dapat mengingat bahwa "dia" sebelumnya adalah subjek kalimat, dan mempertahankan informasi ini dalam cell state. Gate diatur sedemikian rupa agar memori ini tidak dibuang terlalu awal.

5.5 Contoh Implementasi LSTM dengan PyTorch

Berikut ini adalah implementasi model LSTM menggunakan PyTorch untuk memprediksi nilai dari fungsi sinusoidal.

1. Persiapan Data: Deret Sinus

```
import numpy as np
import torch
from torch import nn
import matplotlib.pyplot as plt

# Membuat data sinus
x = np.linspace(0, 100, 1000)
y = np.sin(x)

# Membuat sequence input dan target
def create_sequences(data, seq_len):
    X = []
    Y = []
    for i in range(len(data) - seq_len):
        X.append(data[i:i+seq_len])
        Y.append(data[i+seq_len])
    return np.array(X), np.array(Y)

SEQ_LEN = 50
X, Y = create_sequences(y, SEQ_LEN)

# Tambahkan dimensi batch dan fitur
X = X[..., np.newaxis]
Y = Y[..., np.newaxis]

X_tensor = torch.tensor(X, dtype=torch.float32)
Y_tensor = torch.tensor(Y, dtype=torch.float32)
```

2. Arsitektur LSTM

```
class LSTMModel(nn.Module):
    def __init__(self, input_size=1, hidden_size=50,
        num_layers=1, output_size=1):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size,
            num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        lstm_out, _ = self.lstm(x)  # ignore hidden state
```

```
        output = self.fc(lstm_out[:, -1, :])
        # Ambil output waktu terakhir
        return output

model = LSTMModel()
```

3. Training Model

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

EPOCHS = 30
for epoch in range(EPOCHS):
    output = model(X_tensor)
    loss = criterion(output, Y_tensor)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 5 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

4. Visualisasi Hasil

```
# Prediksi
model.eval()
with torch.no_grad():
    predicted = model(X_tensor).numpy()

plt.figure(figsize=(10,4))
plt.plot(Y.flatten(), label='Asli')
plt.plot(predicted.flatten(), label='Prediksi')
plt.legend()
plt.title("Prediksi Deret Sinus dengan LSTM")
plt.show()
```

Model ini menunjukkan bahwa LSTM mampu menangkap pola berulang dalam data sekuensial dengan memori yang lebih panjang dibandingkan RNN biasa.

5.6 Gated Recurrent Unit (GRU)

Motivasi dan Struktur Dasar

Gated Recurrent Unit (GRU) diperkenalkan oleh Cho et al. (2014) sebagai alternatif yang lebih ringan dari LSTM. GRU memiliki struktur yang lebih sederhana karena hanya

menggunakan dua buah gate: **update gate** (z_t) dan **reset gate** (r_t), serta tidak memiliki cell state terpisah. Semua informasi disimpan langsung dalam hidden state (h_t).

Persamaan Perhitungan GRU

$$
\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right) \\
\tilde{h}_t &= \tanh\left(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h\right) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

Keterangan:

- x_t : input saat t
- h_{t-1} : hidden state sebelumnya
- $\tilde{h}_t$: kandidat hidden state
- z_t : update gate, menentukan seberapa banyak informasi baru akan ditambahkan
- r_t : reset gate, mengatur berapa banyak informasi lama yang dilupakan
- $\odot$: perkalian elemenwise

Perbandingan GRU vs LSTM

GRU dapat digunakan ketika efisiensi komputasi lebih diprioritaskan dan hasil empirisnya cukup baik. Namun, pada tugas dengan dependensi jangka sangat panjang, LSTM masih menjadi pilihan utama.

Implementasi GRU dengan PyTorch

Kita akan menggunakan kembali dataset sinusoidal yang telah digunakan sebelumnya, namun mengganti arsitektur dari LSTM menjadi GRU.

```
class GRUModel(nn.Module):
    def __init__(self, input_size=1,
        hidden_size=50, num_layers=1, output_size=1):
        super(GRUModel, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size,
            num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.gru(x)
        out = self.fc(out[:, -1, :])
        return out

model = GRUModel()
```

Prosedur pelatihan dan evaluasi sama seperti LSTM. Hasil prediksi dari GRU biasanya cukup sebanding dengan LSTM dalam tugas sederhana seperti ini, namun dengan waktu pelatihan yang sedikit lebih cepat.

Fitur	LSTM	GRU
Jumlah gate	Memiliki tiga gerbang utama: *forget*, *input*, dan *output*. Masing-masing mengatur aliran informasi secara eksplisit.	Hanya menggunakan dua gerbang: *update* dan *reset*, membuat struktur lebih sederhana.
Memori eksplisit	Menggunakan cell state terpisah sebagai memori jangka panjang.	Tidak memiliki cell state; hanya hidden state yang digunakan sebagai memori.
Kompleksitas parameter	Lebih kompleks dengan lebih banyak parameter karena memiliki lebih banyak gerbang.	Lebih ringan secara komputasi karena jumlah parameter lebih sedikit.
Waktu pelatihan	Sedikit lebih lambat karena kompleksitas internal.	Lebih cepat dilatih dalam kebanyakan kasus.
Kemampuan modeling sekuens panjang	Sangat baik dalam menangkap dependensi jangka panjang.	Umumnya cukup baik, namun dalam beberapa kasus performa lebih rendah dibanding LSTM.

Tabel 5.1. Perbandingan antara LSTM dan GRU

5.7 Autoencoder

Autoencoder adalah jenis jaringan neural yang dirancang untuk mempelajari representasi (encoding) dari data input, biasanya untuk tujuan reduksi dimensi (serupa dengan PCA) atau *denoising* (mengurangi *noise* pada data). Jaringan ini terdiri dari dua bagian utama:

- **Encoder**: memetakan data input ke representasi berdimensi lebih rendah.
- **Decoder**: merekonstruksi kembali data input dari representasi tersebut.

Arsitektur Autoencoder

Struktur dasar Autoencoder adalah jaringan simetris yang terdiri dari:

- Input layer
- Hidden layer (bottleneck)
- Output layer (berusaha merekonstruksi input)

Gambaran umum arsitektur:

$$\text{Input} \rightarrow \text{Encoder} \rightarrow \text{Bottleneck} \rightarrow \text{Decoder} \rightarrow \text{Reconstruction}$$

Autoencoder dilatih untuk meminimalkan loss antara input dan hasil rekonstruksi, biasanya menggunakan **Mean Squared Error (MSE)**:

$$L = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2$$

Implementasi Autoencoder Sederhana (PyTorch)

Kita akan membuat Autoencoder untuk dataset `MNIST`. Data digit (28x28) akan diencode menjadi vektor berdimensi kecil (32) dan kemudian direkonstruksi.

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

# 1. Load dataset
transform = transforms.ToTensor()
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
                                           transform=transform, download=True)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=Tru

# 2. Definisikan arsitektur Autoencoder
class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 32),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(32, 128),
            nn.ReLU(),
            nn.Linear(128, 28*28),
            nn.Sigmoid(),  # karena pixel [0,1]
            nn.Unflatten(1, (1, 28, 28))
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
```

```
        return x

model = Autoencoder()
```

Training Autoencoder

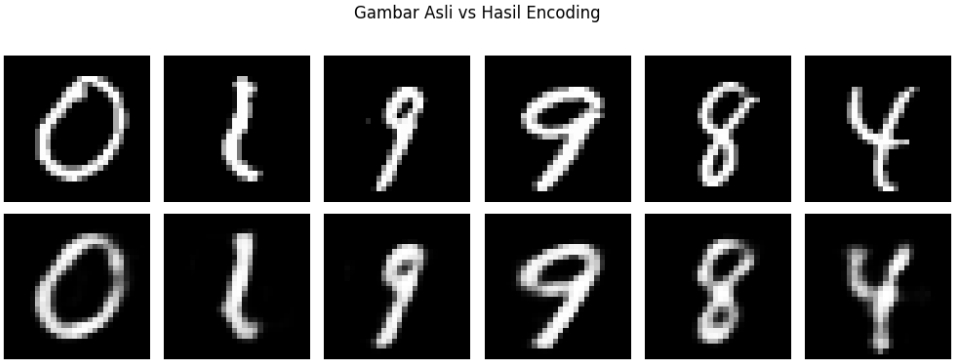
```
# 3. Training
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

for epoch in range(10):
    for img, _ in train_loader:
        output = model(img)
        loss = criterion(output, img)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

Dari hasil training ini, kita telah mendapatkan sebuah model Autoencoder yang dapat mengkodekan/mentransformasikan data asli (gambar ukuran $28 \times 28 = 784$ piksel) menjadi vektor berukuran 32.

Visualisasi Rekonstruksi

```
import matplotlib.pyplot as plt

# Ambil sample dari data
images, _ = next(iter(train_loader))
reconstructed = model(images)

# Plot asli vs hasil rekonstruksi
plt.figure(figsize=(10, 4))
for i in range(6):
    # Original
    plt.subplot(2, 6, i+1)
    plt.imshow(images[i][0].detach().numpy(), cmap='gray')
    plt.axis('off')
    # Reconstructed
    plt.subplot(2, 6, i+7)
    plt.imshow(reconstructed[i][0].detach().numpy(), cmap='gray')
    plt.axis('off')
plt.suptitle("Original vs Reconstructed")
plt.tight_layout()
plt.show()
```

Dari proses di atas, kita akan mendapatkan representasi gambar/citra hasil rekonstruksi dari encoding yang dilakukan oleh Autoencoder yang kita latih. Gambar 5.5



Gambar 5.5. Gambar asli vs hasil rekonstruksi dari Autoencoder

menunjukkan contoh perbandingan gambar asli dibandingkan dengan gambar hasil rekonstruksi dari encoding yang dihasilkan Autoencoder.

Encoding dari Autoencoder dapat digunakan sebagai representasi fitur untuk tugas lain seperti klasifikasi atau clustering. Namun, karena Autoencoder dilatih tanpa label, representasi ini tidak selalu optimal untuk membedakan kelas. Kita akan mempelajari pendekatan yang lebih efektif dalam modul-modul berikutnya.

Referensi dan Bahan Bacaan Lanjutan

- **Neural Networks and Deep Learning by Michael Nielsen** Buku online yang sangat baik untuk pemula, membahas ANN dan backpropagation secara intuitif. http://neuralnetworksanddeeplearning.com/
- **Sklearn Neural Network (MLPClassifier)** Dokumentasi resmi scikit-learn untuk MLP. https://scikit-learn.org/stable/modules/neural_networks_supervised.html
- **3Blue1Brown – Neural Networks Series (YouTube)** Visualisasi konsep backpropagation dan jaringan saraf secara menarik. https://www.3blue1brown.com/topics/neural-networks
- **CS231n – Convolutional Neural Networks for Visual Recognition (Stanford)** Materi kuliah klasik dari Stanford, dengan penjelasan rinci dan gambar. https://cs231n.github.io/convolutional-networks/
- **PyTorch Official – Training a Classifier with CNN** Tutorial resmi membuat CNN dengan PyTorch. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- **Machine Learning Mastery – CNN for Beginners** Panduan praktis untuk membangun CNN dari awal. https://machinelearningmastery.com/
- **Colah's Blog - Understanding LSTM Networks** Penjelasan visual dan intuitif mengenai LSTM oleh Christopher Olah. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- **PyTorch Official Tutorials – Sequence Modeling with LSTM/GRU** Tutorial resmi PyTorch mengenai penggunaan LSTM dan GRU dalam modeling sekuens.
  https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html
- **Deep Learning Book by Ian Goodfellow (Bab 10)** Buku ini menyediakan penjelasan teoretis tentang RNN, LSTM dan GRU.
  https://www.deeplearningbook.org/
- **CS231n - Stanford CNN Course (Winter 2020)** Meski berfokus pada Computer Vision, banyak bahasan tentang arsitektur neural network yang relevan.
  https://cs231n.github.io/
- **Towards Data Science – Blog tentang LSTM dan GRU** Artikel-artikel populer tentang penerapan dan penjelasan LSTM dan GRU.
  https://towardsdatascience.com/

BAB 6
Studi Kasus

Di bab ini, kita akan melihat beberapa contoh penerapan metode-metode *deep learning* pada beberapa permasalahan dan data. Implementasi yang digunakan pada contoh-contoh ini dapat diterapkan dan menjadi dasar bagi penyelesaian masalah-masalah lain yang serupa.

6.1 Studi Kasus: Pengenalan Digit (MNIST) Menggunakan CNN

Dalam studi kasus ini, kita akan membangun model Deep Learning berbasis **Convolutional Neural Network (CNN)** untuk mengklasifikasikan gambar angka tulisan tangan dari dataset **MNIST**.

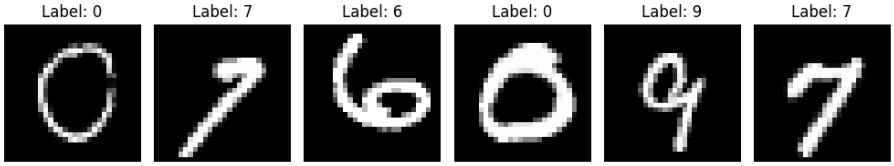
Dataset ini merupakan salah satu dataset benchmark paling terkenal untuk pengenalan citra sederhana. MNIST berisi:

- 60.000 gambar untuk pelatihan dan 10.000 gambar untuk pengujian.
- Setiap gambar berukuran 28x28 piksel dalam skala abu-abu (grayscale).
- Label berupa angka dari 0 hingga 9.

1. Memuat Dataset dan Menampilkan Contoh

MNIST tersedia secara otomatis di pustaka `torchvision`, sehingga kita tidak perlu mengunduh secara manual. Dataset akan secara otomatis disimpan di folder `./data` setelah pertama kali diunduh.

Berikut adalah kode untuk memuat data dan menampilkan beberapa contoh gambar. Kode Python berikut akan melakukan pengunduhan dan *load* (memuat) data MNIST. Gambar 6.1 menunjukkan contoh gambar pada data MNIST.



Gambar 6.1. Contoh gambar pada data MNIST

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Transformasi ke tensor
transform = transforms.ToTensor()

# Dataset MNIST
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Tampilkan beberapa sampel
examples = enumerate(trainloader)
batch_idx, (example_data, example_targets) = next(examples)

plt.figure(figsize=(10, 2))
for i in range(6):
    plt.subplot(1,6,i+1)
    plt.imshow(example_data[i][0], cmap='gray')
    plt.title(f"Label: {example_targets[i].item()}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

2. Arsitektur CNN Sederhana

Arsitektur CNN yang digunakan dalam studi kasus ini memiliki struktur sebagai berikut:

- **Conv1**: Layer konvolusi dengan 10 filter berukuran 5x5.
- **MaxPool1**: Operasi `max pooling` dengan ukuran 2x2.
- **Conv2**: Layer konvolusi kedua dengan 20 filter berukuran 5x5.
- **MaxPool2**: Max pooling 2x2 kedua.
- **FC1 (Linear)**: Fully connected layer dengan 50 neuron.
- **FC2 (Output)**: Fully connected layer dengan 10 output (untuk kelas 0 hingga 9).

Arsitektur ini cukup sederhana namun efektif untuk tugas klasifikasi citra kecil seperti MNIST.

Kode Python:

```
import torch.nn as nn
```

```
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)  # flatten
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleCNN()
```

3. Pelatihan dan Evaluasi Model

Langkah selanjutnya adalah melatih model menggunakan data pelatihan dan mengevaluasinya pada data pengujian.

- Optimizer: `Adam`
- Loss Function: `CrossEntropyLoss`, cocok untuk klasifikasi multi-kelas
- Epoch: 5 (dapat ditambah sesuai kebutuhan)

Perhatikan bahwa kode pelatihan di bawah ini idealnya dijalankan pada sebuah mesin dengan kemampuan GPU yang sesuai, agar tidak membutuhkan waktu terlalu lama. Pada mesin GPU jenis T4 di Google Colab, misalnya, kode pelatihan di bawah membutuhkan waktu sekitar 2 menit. **Kode Python:**

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Loop pelatihan
for epoch in range(5):
    model.train()
    running_loss = 0.0
    for inputs, labels in trainloader:
        optimizer.zero_grad()
        outputs = model(inputs)
```

```
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader):.4f}")
```

Apabila dijalankan, kode di atas akan menghasilkan keluaran seperti dicontohkan di bawah ini:

```
Epoch 1, Loss: 0.2898
Epoch 2, Loss: 0.0798
Epoch 3, Loss: 0.0547
Epoch 4, Loss: 0.0426
Epoch 5, Loss: 0.0358
```

Keluaran in menunjukkan bahwa dalam 5 epoch, model CNN yang dibangun berhasil mempelajari data latih dan mendapatkan error *cross-entropy* yang cukup kecil (0.0358). Untuk dapat menunjukkan bahwa model telah berhasil menangkap pola umum dari data, kita coba lakukan pengujian pada model dengan menggunakan data uji yang belum pernah dipakai pada data latih.

Evaluasi Akurasi pada Data Uji:

```
# Evaluasi performa pada test set
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Akurasi pada data uji: {100 * correct / total:.2f}%")
```

Kode pengujian di atas mungkin akan menghasilkan keluaran sebagai berikut:

```
Akurasi pada data uji: 98.35%
```

Kesimpulan

Model CNN sederhana ini cukup efektif untuk tugas klasifikasi gambar MNIST, bahkan dengan arsitektur minimal, model dapat mencapai akurasi lebih dari 98% pada data uji hanya dalam beberapa epoch.

Latihan ini memperkenalkan pipeline dasar Deep Learning untuk Computer Vision, termasuk:

- Penggunaan dataset populer seperti MNIST
- Desain arsitektur CNN
- Proses training dan evaluasi dengan PyTorch

Gambar 6.2. Contoh gambar bunga pada dataset Oxford 102 Flowers

6.2 Studi Kasus: Klasifikasi Citra Bunga Menggunakan *Pre-Trained Model & Fine-Tuning*

Pendahuluan

Pada studi kasus sebelumnya (MNIST), kita dapat membangun arsitektur CNN dari awal karena datasetnya relatif sederhana: gambar grayscale 28×28 piksel dan hanya memiliki 10 kelas.

Namun, ketika kita berhadapan dengan data berwarna (RGB) beresolusi tinggi dan memiliki banyak kelas seperti pada dataset Oxford 102 Flower (102 kelas bunga) yang akan kita gunakan disini, membangun CNN dari nol bisa sangat sulit dan tidak efisien. Contoh gambar bunga pada dataset ini dapat dilihat pada Gambar 6.2

Untuk itu, kita gunakan pendekatan transfer learning: memanfaatkan model pralatih (*pretrained*) seperti `ResNet18` yang sudah dilatih di ImageNet dan menyesuaikannya untuk tugas baru ini. Kelebihan dari pendekatan ini adalah, kita tidak perlu melatih dengan data kita sendiri dari awal, melainkan kita dapat menggunakan model yang sudah terlatih (mungkin untuk pengenalan citra yang lain), dan kemudian melatihnya lagi lebih lanjut (*fine-tune*) untuk permasalahan yang ada di data kita. Kita bisa melakukan ini dengan:

Freeze sebagian besar layer awal (fitur umum) **Fine-tune** layer akhir (khusus tugas/permasalahan kita)

Menyiapkan Dataset dan Struktur Folder

Dataset Oxford 102 Flower dapat diunduh dari:
https://www.robots.ox.ac.uk/~vgg/data/flowers/102/

Dataset ini terdiri dari 102 kategori bunga, masing-masing berisi antara 40–250 gambar. Untuk memudahkan persiapan data, kita bisa download data tersebut, beserta label dan pembagian (*split*) data secara langsung dari web dengan menggunakan perintah berikut pada *notebook* Python kita.

```
!wget
https://www.robots.ox.ac.uk/~vgg/data/flowers/102/102flowers.tgz
!wget
https://www.robots.ox.ac.uk/~vgg/data/flowers/102/imagelabels.mat
!wget
https://www.robots.ox.ac.uk/~vgg/data/flowers/102/setid.mat
```

Setelah itu, kita ekstrak data dengan menggunakan perintah

```
!tar xfz 102flowers.tgz
```

Setelah itu kita akan dapatkan sebuah folder baru bernama `jpg` berisi semua gambar-gambar pada dataset Oxford Flower 102. Selanjutnya kita lakukan pelabelan dan pemisahan data menjadi data latih dan data validasi dengan menggunakan program berikut.

```
import scipy.io
import os
import shutil
from pathlib import Path
from PIL import Image

# Path ke file dan gambar
img_dir = Path("jpg")
labels = scipy.io.loadmat("imagelabels.mat")["labels"][0]
setid = scipy.io.loadmat("setid.mat")

# Konversi 1-based indexing MATLAB ke 0-based Python
train_ids = setid["trnid"][0] - 1
val_ids = setid["valid"][0] - 1
test_ids = setid["tstid"][0] - 1

# Semua ID dan label
all_ids = list(range(len(labels)))

# Buat struktur direktori
for split, ids in [("train", train_ids), ("val", val_ids)]:
    for idx in ids:
        label = int(labels[idx])
        src = img_dir / f"image_{idx+1:05d}.jpg"
        dst_dir = Path(f"data/flowers/{split}/class_{label:03d}")
        dst_dir.mkdir(parents=True, exist_ok=True)
        shutil.copy(src, dst_dir / src.name)
```

Dari hasil pemanggilan kode di atas, maka kita akan mendapatkan sebuah folder baru bernama `data/flowers` berisi semua file gambar yang ada di dataset tersebut, dan telah dipisahkan menjadi data latih serta validasi, serta dilabeli dengan nama kelasnya.

Melakukan Pelatihan Model

Setelah data siap untuk digunakan, kita lakukan langkah-langkah untuk melatih dan *fine-tune* model *pre-trained* kita. Dalam hal ini, kita akan menggunakan Resnet18 yang tergolong dalam kelompok *residual deep learning network*. Resnet18 memiliki 18 layer, 17 yang pertama adalah *convolutional layer*, sedangkan layer terakhir adalah *fully-connected*. Resnet18 yang disediakan di pustaka Pytorch telah dilatih menggunakan dataset ImageNet (https://www.image-net.org/). Oleh karena itu, model ini memiliki kemampuan untuk menangkap berbagai fitur citra/gambar yang penting dan dapat digunakan pada proses *transfer learning* untuk menyelesaikan permasalahan klasifikasi citra yang lain.

Pertama, kita buat fungsi untuk melakukan transformasi pada data yang akan diperlukan oleh Resnet18.

```
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torchvision import models
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Transformasi untuk data training (augmentasi) dan validasi
transform_train = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

transform_val = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

Kode di atas menyiapkan dua buah fungsi transformasi untuk data latih dan data validasi. Berikut penjelasan detail dari kedua fungsi transformasi tersebut.

1. `transforms.RandomResizedCrop(224)` Memotong bagian acak dari gambar asli dan me-resize hasil potongan tersebut menjadi ukuran 224×224 piksel. Tujuannya: menambah variasi input dan membuat model lebih robust terhadap posisi objek.
2. `transforms.RandomHorizontalFlip()` Membalik gambar secara horizontal dengan probabilitas 0.5. Augmentasi ini membantu model tidak terlalu bergantung pada orientasi objek (misalnya, bunga bisa menghadap kiri atau kanan).
3. `transforms.ToTensor()` Mengubah gambar PIL (Python Imaging Library) menjadi tensor PyTorch dengan ukuran [C, H, W] dan skala nilai piksel dari [0, 255] ke [0.0, 1.0].
4. `transforms.Normalize(mean, std)` Melakukan normalisasi channel-wise (R, G, B) menggunakan nilai rata-rata (mean) dan standar deviasi (std) tertentu. Nilai-nilai `mean` dan `std` yang digunakan disini didapatkan dari data latih ImageNet yang telah digunakan untuk melatih Resnet18.
5. Untuk `transform_val` yang akan diterapkan pada data validasi, tidak ada augmentasi karena data validasi harus sesuai kondisi aslinya, namun tetap dilakukan resize dan normalisasi.

Selanjutnya kita menyiapkan data untuk pelatihan dan mengambil model Resnet18 yang sudah *pre-trained* dari Pytorch. Pastikan untuk menjalankan kode ini pada mesin dengan kemampuan GPU yang mencukupi.

```
# Load dataset training dan validasi
train_dataset = ImageFolder('data/flowers/train', transform=transform_train)
val_dataset = ImageFolder('data/flowers/val', transform=transform_val)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

# Cek jumlah kelas
print(f"Jumlah kelas: {len(train_dataset.classes)}")

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = models.resnet18(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 102)  # Oxford 102 = 102 kelas
model = model.to(device)
```

Selanjutnya kita definisikan fungsi untuk pelatihan (*fine-tuning)*dan validasi model.

```
# Definisi fungsi loss dan optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Fungsi training
def train_one_epoch(model, loader, optimizer, criterion):
```

```
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, preds = torch.max(outputs, 1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    acc = 100 * correct / total
    print(f"Train Loss: {running_loss:.4f}, Accuracy: {acc:.2f}%")

# Fungsi validasi
def validate(model, loader, criterion):
    model.eval()
    total_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_loss += loss.item()

            _, preds = torch.max(outputs, 1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)

    acc = 100 * correct / total
    print(f"Val Loss: {total_loss:.4f}, Accuracy: {acc:.2f}%")
```

Setelah semua siap, kita lakukan pelatihan.

```
# Pelatihan selama beberapa epoch
for epoch in range(10):
# bisa diganti jadi 15-20 untuk hasil lebih baik
```

```
    print(f"Epoch {epoch+1}")
    train_one_epoch(model, train_loader, optimizer, criterion)
    validate(model, val_loader, criterion)
```

Setelah kita jalankan proses pelatihan di atas, model akan mempelajari klasifikasi pada data baru ini, dan progress-nya dapat kita amati pada setiap epoch. Contoh hasil yang kita dapatkan mungkin seperti ini:

```
..... [hanya menampilkan 5 epoch terakhir]
Epoch 11
Train Loss: 32.6415, Accuracy: 74.22%
Val Loss: 61.4957, Accuracy: 53.53%
Epoch 12
Train Loss: 29.0871, Accuracy: 76.67%
Val Loss: 58.6292, Accuracy: 54.12%
Epoch 13
Train Loss: 25.5528, Accuracy: 78.14%
Val Loss: 71.5673, Accuracy: 47.94%
Epoch 14
Train Loss: 21.5676, Accuracy: 83.14%
Val Loss: 46.9770, Accuracy: 62.55%
Epoch 15
Train Loss: 21.4556, Accuracy: 82.16%
Val Loss: 55.5522, Accuracy: 59.41%
```

Dari hasil di atas, kita bisa mengamati:

- Akurasi pelatihan meningkat stabil dari 74% ke 83% — ini menunjukkan model belajar dari data.
- Namun, akurasi validasi cukup fluktuatif, dan bahkan sempat turun (epoch 13) — ini mengindikasikan potensi overfitting.
- Akurasi validasi terbaik terjadi di epoch 14 (62.55%), yang cukup baik untuk awal, tapi masih banyak ruang untuk peningkatan.

5. Evaluasi dan Interpretasi Hasil

Setelah proses pelatihan selesai, langkah berikutnya adalah melakukan evaluasi model terhadap data validasi. Kita tidak hanya melihat akurasi, tetapi juga ingin memahami bagaimana model memprediksi masing-masing kelas.

Berikut adalah cuplikan kode Python untuk menyimpan model terbaik (dengan akurasi validasi tertinggi):

```
best_acc = 0.0

for epoch in range(epochs):
    train_loss, train_acc = train_model(...)
    val_loss, val_acc = evaluate_model(...)
```

```
    if val_acc > best_acc:
        best_acc = val_acc
        torch.save(model.state_dict(), "best_model.pth")
        print(f"Model saved with accuracy: {val_acc:.4f}")
```

Setelah model terbaik disimpan, kita dapat melakukan evaluasi yang lebih rinci seperti classification report dan confusion matrix.

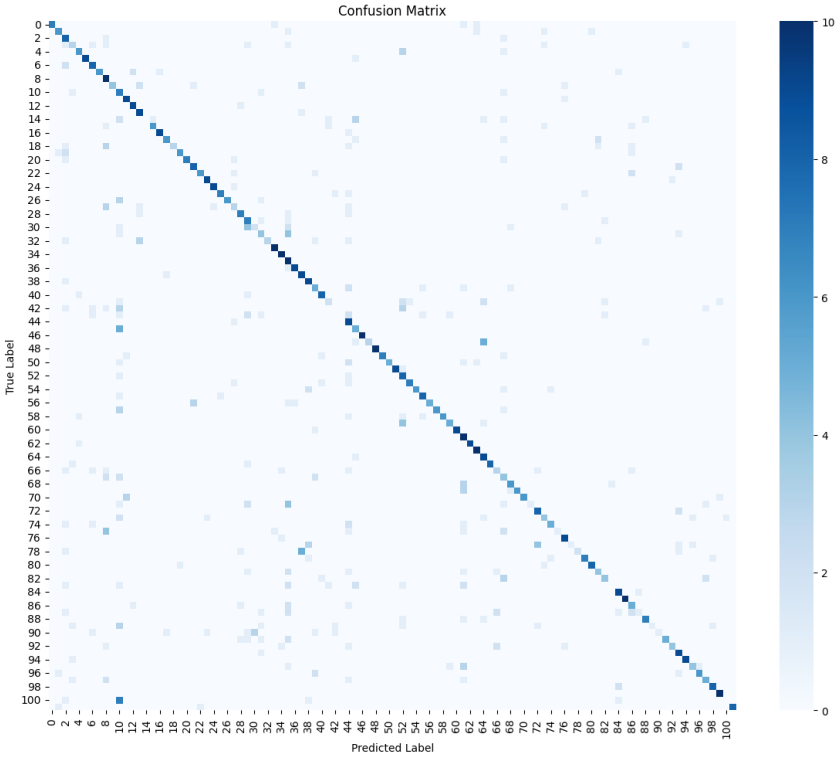
```
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Inference pada data validasi
model.eval()
all_preds = []
all_labels = []

with torch.no_grad():
    for images, labels in val_loader:
        images = images.to(device)
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.numpy())

# Report klasifikasi
print(classification_report(all_labels, all_preds))

# Confusion Matrix
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(12, 10))
sns.heatmap(cm, annot=False, cmap='Blues', fmt='d')
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.tight_layout()
plt.show()
```

Kode di atas akan menampilkan hasil seperti di bawah ini.

```
    accuracy                           0.61      1020
   macro avg       0.70      0.61      0.60      1020
weighted avg       0.70      0.61      0.60      1020
```

yang menunjukkan akurasi sekitar 60% pada data validasi, dengan *confusion matrix* seperti ditunjukkan pada Gambar 6.3.

Catatan: Report klasifikasi akan memberikan metrik seperti precision, recall, dan F1-score untuk masing-masing kelas, sedangkan confusion matrix membantu kita memahami kelas mana yang sering keliru diprediksi.



Gambar 6.3. Confusion Matrix hasil pengujian Resnet18 pada dataset Oxford Flower 102

Analisis Hasil dan Saran Peningkatan

Model pretrained seperti ResNet-18 sudah cukup baik memberikan akurasi validasi > 60% dalam 10 - 15 epoch. Namun demikian, masih banyak ruang untuk peningkatan, misalnya:

- **Unfreeze beberapa layer awal** secara bertahap (layer-wise unfreezing) agar model dapat menyesuaikan diri lebih baik dengan domain bunga.
- **Data augmentasi tambahan** seperti random rotation, color jitter, dan brightness adjustment.
- **Penyesuaian learning rate**, misalnya dengan `torch.optim.lr_scheduler` untuk decay otomatis.
- **Transfer learning dari model lebih besar**, seperti ResNet50 atau EfficientNet, jika sumber daya memadai.

Eksperimen lebih lanjut sangat disarankan untuk mencapai performa yang optimal.

6.3 Studi Kasus: Analisis Sentimen dengan LSTM

Dalam studi kasus ini, kita akan melatih model LSTM sederhana untuk melakukan analisis sentimen terhadap teks ulasan. Model akan menerima teks sebagai input dan memprediksi apakah ulasan tersebut bersentimen positif atau negatif. Kita menggunakan dataset IMDB (ulasan film) yang sudah cukup dikenal untuk tugas ini.

1. Persiapan dan Pembacaan Dataset

Dataset IMDB dapat diunduh dari berbagai sumber (misalnya melalui `keras.datasets`). Namun di sini kita akan menggunakan versi CSV yang tersedia secara publik dan lebih mudah diproses. Data ini dapat didownload misalnya dari:
https://huggingface.co/datasets/stanfordnlp/imdb

Dataset tersebut berisi dua kolom, yaitu kolom `review` (teks ulasan dalam bahasa Inggris) dan kolom dan `sentiment` (bernilai 'positive' atau 'negative'). Gambar 6.4 menunjukkan beberapa contoh isi data tersebut. Setelah mendownload dan menempatkan file CSV ke lokasi yang sesuai, gunakan perintah berikut untuk membaca data.

```
import pandas as pd
import numpy as np
import torch
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
df = pd.read_csv("imdb_sentiment.csv")  # pastikan file ada di path ini
print(df.head())
```

review	sentiment
One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be h	positive
A wonderful little production. The filming technique is very unassuming- very o	positive
I thought this was a wonderful way to spend time on a too hot summer weekend, sitting in	positive
Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & hi	negative
Petter Mattei's "Love in the Time of Money" is a visually stunning film to watch. Mr. Mattei	positive
Probably my all-time favorite movie, a story of selflessness, sacrifice and dedication to a r	positive
I sure would like to see a resurrection of a up dated Seahunt series with the tech they have	positive
This show was an amazing, fresh & innovative idea in the 70's when it first aired. The first 7	negative
Encouraged by the positive comments about this film on here I was looking forward to wat	negative
If you like original gut wrenching laughter you will like this movie. If you are young or old the	positive

Gambar 6.4. Contoh data IMDB Review

2. Tokenisasi dan Padding

Tokenisasi adalah tahapan praproses data teks yang digunakan untuk memecah teks menjadi unit-unit penyusun bahasa alami, misalnya kata atau frasa. Kita akan menggunakan tokenizer dari `keras.preprocessing.text` untuk mengubah teks menjadi urutan token, kemudian melakukan padding agar panjang semua input sama.

```
from keras.preprocessing.text import Tokenizer
from keras.utils import pad_sequences

# Tokenisasi
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(df['text'])
sequences = tokenizer.texts_to_sequences(df['text'])

# Padding
X = pad_sequences(sequences, maxlen=200, padding='post', truncating='post')
y = df['label'].values
```

3. Split Data dan Konversi ke Tensor

Gunakan data loader dari `torch` untuk mengkonversi data menjadi bentuk tensor.

```
from torch.utils.data import DataLoader, TensorDataset

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Konversi ke tensor PyTorch
X_train_tensor = torch.tensor(X_train, dtype=torch.long)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.long)
```

```
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Dataloader
train_ds = TensorDataset(X_train_tensor, y_train_tensor)
test_ds = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_ds, batch_size=64, shuffle=True)
test_loader = DataLoader(test_ds, batch_size=64)
```

4. Arsitektur LSTM Sederhana

Untuk membangun model klasifikasi sentimen berbasis teks, kita menggunakan arsitektur jaringan saraf LSTM. Model ini terdiri dari tiga komponen utama:

- Pertama, lapisan `Embedding` digunakan untuk mengubah token kata menjadi representasi vektor berdimensi tetap.Embedding adalah teknik ekstraksi fitur dalam pengolahan bahasa alami (NLP) yang digunakan untuk mengubah data teks menjadi representasi numerik (vektor). Kita akan mempelajari teknik embedding ini pada modul berikutnya. Disini, kita cukup menggunakan teknik embedding sederhana yang disediakan oleh `Pytorch`, yaitu `nn.Embedding`.
- Kedua, vektor-vektor tersebut diproses oleh lapisan `LSTM` untuk menangkap pola berurutan dan konteks dalam kalimat.
- Terakhir, keluaran dari LSTM dimasukkan ke lapisan `Linear` (*fully connected*), diikuti oleh fungsi aktivasi sigmoid untuk menghasilkan probabilitas prediksi kelas sentimen (positif atau negatif).

```
import torch.nn as nn

class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        embedded = self.embedding(x)
        _, (hidden, _) = self.lstm(embedded)
        out = self.fc(hidden[-1])
        return self.sigmoid(out).squeeze()
```

5. Pelatihan Model

Selanjutnya, kita definisikan proses pelatihan sebagai berikut:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
model = SentimentLSTM(vocab_size=10000, embedding_dim=128,
        hidden_dim=128).to(device)
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

# Training loop
for epoch in range(10):
    model.train()
    total_loss = 0
    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)
        preds = model(xb)
        loss = loss_fn(preds, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss/len(train_loader):.4f}")
```

Kita mungkin akan mendapatkan hasil pelatihan sebagai berikut:

```
Epoch 1, Loss: 0.6934
Epoch 2, Loss: 0.6868
Epoch 3, Loss: 0.5778
Epoch 4, Loss: 0.3628
Epoch 5, Loss: 0.2767
Epoch 6, Loss: 0.2304
Epoch 7, Loss: 0.1938
Epoch 8, Loss: 0.1711
Epoch 9, Loss: 0.1373
Epoch 10, Loss: 0.1252
```

Hasil ini menunjukkan bahwa model berhasil mempelajari pola pada data dan menurunkan *loss* pada pelatihan.

6. Evaluasi Model

Setelah model dilatih, kita lakukan evaluasi menggunakan data uji yang sudah kita siapkan.

```
from sklearn.metrics import accuracy_score, classification_report

model.eval()
all_preds, all_labels = [], []

with torch.no_grad():
    for xb, yb in test_loader:
```

```
        xb = xb.to(device)
        preds = model(xb).cpu().numpy() > 0.5
        all_preds.extend(preds)
        all_labels.extend(yb.numpy())

acc = accuracy_score(all_labels, all_preds)
print("Test Accuracy:", acc)
print(classification_report(all_labels, all_preds))
```

Kita mungkin akan mendapatkan hasil yang serupa dengan keluaran berikut:

```
Test Accuracy: 0.8695
              precision    recall  f1-score   support

         0.0       0.86      0.88      0.87      5000
         1.0       0.88      0.86      0.87      5000

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.87     10000
weighted avg       0.87      0.87      0.87     10000
```

Hasil ini menunjukkan bahwa model dapat memprediksi dengan akurasi yang lumayan baik, yaitu sekitar 87%.

Catatan

Studi kasus ini menunjukkan bahwa model LSTM sederhana sudah dapat bekerja cukup baik dalam tugas klasifikasi teks biner. Kita bisa meningkatkan hasil lebih jauh dengan:

- Menambahkan dropout atau lapisan tambahan
- Menggunakan embedding yang lebih lanjut, seperti Word2Vec, GloVe dan lain-lain (akan dipelajari kemudian)
- Menggunakan arsitektur RNN yang lebih kompleks (GRU, bidirectional LSTM)
- Menggunakan teknik dan model yang lebih lanjut, seperti transformer, yang akan dipelajari kemudian.

Bibliografi

[DBCR19] David M Diez, Christopher D Barr, and Mine Cetinkaya-Rundel, *Openintro statistics*, 3rd ed., OpenIntro, 2019.

[Dev23] Google Developers, *Machine learning crash course*, https://developers.google.com/machine-learning/crash-course, 2023.

[DFO20] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong, *Mathematics for machine learning*, Cambridge University Press, 2020.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, Book in preparation for MIT Press.

[HTF01] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *The elements of statistical learning*, Springer Series in Statistics, Springer New York Inc., New York, NY, USA, 2001.

[ID17] Barbara Illowsky and Susan Dean, *Introductory statistics*, OpenStax, 2017.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, *An introduction to statistical learning*, Springer, 2013.

[Uni23] Duke University, *Data science math skills*, https://www.coursera.org/learn/datasciencemathskills, 2023.

[Was04] Larry Wasserman, *All of statistics: A concise course in statistical inference*, Springer, 2004.