Deep Learning Cheatsheet : Panduan Lengkap untuk Pemula

Disusun oleh Tim Datasans
https://datasans.medium.com/

Peringatan

Materi ini telah divalidasi, namun bagaimanapun juga, ebook ini tidak luput dari kesalahan baik definisi, konten secara umum, maupun syntax. Segala masukkan dari pengguna sangat terbuka. DM kami di instagram @datasans.book

Himbauan

1. Tidak menjadikan ebook ini satu-satunya sumber pegangan, *cross check* dan validasi segala informasi dari sumber lain.
2. **Tidak membagikan atau mencetaknya untuk diperbanyak**, kecuali untuk pribadi.
3. Disarankan untuk merekomendasikan langsung ke instagram @datasans.book jika temanmu berminat agar ilmu yang bermanfaat bisa tersebar semakin luas.

Daftar Isi

Bab 1. Pendahuluan

1.1 Pengertian Deep Learning

Deep learning adalah cabang dari kecerdasan buatan (AI) yang menggunakan model matematika yang disebut neural networks. Mengapa disebut 'deep'? Karena model ini menggunakan banyak lapisan proses pemrosesan data, atau apa yang kita kenal dengan 'layers'. Setiap layer berisi node yang melakukan serangkaian kalkulasi matematis sederhana, dan masing-masing layer belajar untuk mengenali fitur yang berbeda dari input yang diberikan.

Deep learning bisa dipahami sebagai proses pengekstrakan fitur otomatis dari data dan menggunakannya untuk melakukan prediksi atau keputusan. Ini sangat berbeda dari metode pembelajaran mesin tradisional, di mana fitur harus didefinisikan secara manual dan kemudian dikenakan pada model pembelajaran mesin.

1.2 Sejarah dan Perkembangan Deep Learning

Konsep dari neural networks bukanlah ide baru. Sebenarnya, ide tersebut telah ada sejak tahun 1940-an. Namun, deep learning sejati baru berjalan ketika kita memiliki komputasi yang cukup kuat untuk melatih jaringan saraf yang 'dalam'.

Pada dekade 1980-an dan 1990-an, peneliti mulai bereksperimen dengan jaringan yang lebih dalam, tetapi mereka menemui kendala dalam pelatihan model tersebut. Namun, penemuan kembali algoritma backpropagation pada tahun 1986 memberikan solusi atas masalah ini, memungkinkan gradient dari cost function dapat dihitung secara efisien dan digunakan untuk memperbarui bobot dalam jaringan.

Perkembangan signifikan berikutnya datang pada tahun 2012, ketika tim peneliti dari University of Toronto memenangkan kompetisi ImageNet menggunakan model deep learning, Convolutional Neural Network (CNN). Prestasi ini membuktikan keunggulan deep learning dalam tugas-tugas yang melibatkan pengolahan gambar.

Sejak saat itu, deep learning telah menjadi bagian integral dari berbagai aplikasi dan industri. Kemampuan untuk memproses dan memahami data besar dalam waktu singkat telah mengubah cara kita berinteraksi dengan teknologi dan dunia sekitar kita.

1.3 Aplikasi Deep Learning dalam Kehidupan Sehari-hari

Deep learning tidak lagi terbatas pada laboratorium penelitian atau aplikasi industri spesifik. Sekarang, aplikasi deep learning bisa ditemukan di berbagai aspek kehidupan sehari-hari.

Pengenalan Suara: Kita bisa melihat penggunaan deep learning dalam asisten virtual seperti Google Assistant, Siri, atau Alexa yang menggunakan pengenalan suara untuk menerima dan memahami perintah.

Pengenalan Gambar: Deep learning digunakan dalam teknologi pengenalan wajah yang digunakan oleh Facebook untuk menandai foto atau oleh Apple dalam sistem Face ID.

Penerjemahan Bahasa: Google Translate menggunakan model deep learning yang dikenal sebagai neural machine translation untuk menerjemahkan antar bahasa.

Rekomendasi Produk: Deep learning juga digunakan oleh perusahaan seperti Amazon dan Netflix untuk memberikan rekomendasi produk atau film yang dipersonalisasi.

Kendaraan Otonom: Deep learning memainkan peran penting dalam pengembangan sistem pengendalian otomatis untuk kendaraan otonom, memungkinkan mereka untuk mengenali dan bereaksi terhadap lingkungan sekitarnya.

Pendahuluan ini memberi kita gambaran tentang apa itu deep learning, bagaimana perkembangannya, dan bagaimana kita menggunakannya dalam kehidupan sehari-hari. Selanjutnya, kita akan membahas secara mendalam tentang konsep-konsep penting dalam deep learning dan bagaimana menerapkannya.

Bab 2. Pengenalan Konsep Dasar Deep Learning

2.1 Neural Networks

Neural Networks (Jaringan Saraf) merupakan pondasi utama dari deep learning. Ide dasarnya diambil dari cara kerja otak manusia dalam memproses informasi. Berikut adalah penjelasan singkat dan padat tentang konsep ini:

Struktur Neural Network
Sebuah Neural Network terdiri dari tiga bagian utama:

1. **Input Layer:** Layer ini menerima input data. Setiap neuron dalam layer ini mewakili satu fitur dari data.
2. **Hidden Layer(s):** Layer ini berada di antara input dan output layer. Jumlah hidden layer dan jumlah neuron dalam setiap layer dapat bervariasi, tergantung pada kompleksitas masalah yang sedang dikerjakan.
3. **Output Layer:** Layer ini memberikan hasil akhir dari neural network. Jumlah neuron dalam layer ini biasanya sama dengan jumlah kelas dalam masalah klasifikasi, atau satu neuron untuk masalah regresi.

Setiap neuron dalam satu layer terhubung dengan semua neuron di layer selanjutnya. Hubungan ini diberi bobot, yang di-adjust selama proses pelatihan neural network.

Forward Propagation
Ini adalah langkah pertama dalam proses belajar sebuah neural network. Dalam tahap ini, data dikirimkan melalui jaringan dari input layer ke output layer. Setiap neuron mengambil input, mengalikan dengan bobot, menambahkan bias (sebuah nilai konstan untuk memastikan output tidak nol ketika semua input nol), dan kemudian meneruskan hasil melalui fungsi aktivasi.

Fungsi Aktivasi
Fungsi aktivasi bertugas mengubah input neuron menjadi output yang akan diteruskan ke neuron selanjutnya. Ada beberapa jenis fungsi aktivasi, seperti sigmoid, ReLU (Rectified Linear Unit), dan tanh. Pilihan fungsi aktivasi tergantung pada masalah yang sedang dihadapi dan struktur dari neural network itu sendiri.

Backward Propagation dan Penyesuaian Bobot
Setelah forward propagation, neural network akan menghasilkan output. Output ini kemudian dibandingkan dengan output sebenarnya untuk menghitung kesalahan. Kesalahan ini kemudian dipropagasi mundur melalui jaringan (backward propagation), dimana bobot antar neuron di-adjust berdasarkan kesalahan tersebut. Proses ini

berulang-ulang hingga kesalahan mencapai nilai minimum atau setelah sejumlah iterasi tertentu.

Learning Rate
Learning rate adalah parameter yang mengontrol seberapa cepat atau lambat kita ingin model kita belajar. Nilai yang sangat kecil dapat membuat proses belajar menjadi sangat lambat, sedangkan nilai yang terlalu besar bisa menyebabkan model melewatkan minimum global dari fungsi kesalahan.

Berikut adalah diagram yang menjelaskan struktur dan proses kerja Neural Network:

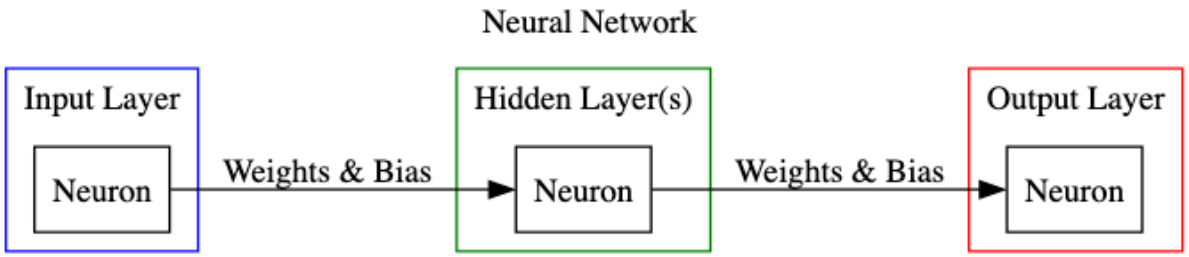


Diagram ini menunjukkan tiga bagian utama dari Neural Network:

1. **Input Layer (Biru):** Layer ini menerima input data. Setiap neuron dalam layer ini mewakili satu fitur dari data.
2. **Hidden Layer(s) (Hijau):** Layer ini berada di antara input dan output layer. Jumlah hidden layer dan jumlah neuron dalam setiap layer dapat bervariasi, tergantung pada kompleksitas masalah yang sedang dikerjakan.
3. **Output Layer (Merah):** Layer ini memberikan hasil akhir dari neural network. Jumlah neuron dalam layer ini biasanya sama dengan jumlah kelas dalam masalah klasifikasi, atau satu neuron untuk masalah regresi.

Setiap neuron dalam satu layer terhubung dengan semua neuron di layer selanjutnya. Hubungan ini diberi bobot, yang di-adjust selama proses pelatihan neural network.

Itulah sekilas tentang konsep dasar Neural Networks. Selanjutnya, kita akan membahas lebih detail tentang fungsi aktivasi, fungsi biaya, backpropagation, dan algoritma optimasi.

2.2 Activation Functions

Fungsi aktivasi memainkan peran penting dalam arsitektur Neural Network. Fungsi ini bertugas untuk menentukan output neuron berdasarkan set input tertentu. Berikut ini adalah ringkasan tentang beberapa fungsi aktivasi yang umum digunakan.

Sigmoid

Sigmoid adalah fungsi aktivasi yang paling sering digunakan. Output dari fungsi ini berkisar antara 0 hingga 1, yang dapat digunakan untuk mengubah angka-arbitrer menjadi probabilitas.

Formula: $\sigma(x) = 1 / (1 + \exp(-x))$

Keuntungan: Output berkisar antara 0 dan 1, berguna untuk mengubah nilai menjadi probabilitas.
Kekurangan: Sigmoid menderita dari apa yang dikenal sebagai "vanishing gradient problem", dimana untuk nilai input yang sangat besar atau sangat kecil, gradiennya hampir nol. Ini menyulitkan proses pembelajaran.

Tanh
Tanh adalah alternatif lain untuk sigmoid. Output dari fungsi ini berkisar antara -1 hingga 1, yang menjadikannya lebih baik dalam menangani negatif input.

Formula: $\tanh(x) = 2\sigma(2x) - 1$

Keuntungan: Sama seperti sigmoid, tetapi lebih baik menangani input negatif.
Kekurangan: Masih menderita vanishing gradient problem.

ReLU (Rectified Linear Unit)
ReLU adalah fungsi aktivasi yang paling umum digunakan di dalam hidden layers dari sebuah neural network. Fungsi ini hanya akan menghasilkan output jika inputnya positif.

Formula: $\text{ReLU}(x) = \max(0, x)$

Keuntungan: ReLU sangat efisien secara komputasi dan tidak menderita vanishing gradient problem.
Kekurangan: ReLU memiliki apa yang dikenal sebagai "dying ReLU problem" - jika neuron memberikan output negatif, neuron tersebut akan "mati", atau berhenti belajar.

Leaky ReLU
Ini adalah versi modifikasi dari ReLU yang mencoba untuk memecahkan masalah dying ReLU dengan memperkenalkan nilai kecil positif untuk input negatif.

Formula: $\text{Leaky ReLU}(x) = \max(0.01x, x)$

Keuntungan: Mengatasi masalah dying ReLU.
Kekurangan: Hasilnya mungkin tidak stabil untuk data berisik.

Softmax
Fungsi Softmax biasanya digunakan di layer output untuk klasifikasi multi-kelas. Fungsi ini akan mengubah vektor input kedalam vektor probabilitas.

Keuntungan: Berguna untuk klasifikasi multi-kelas.
Kekurangan: Tidak cocok untuk digunakan di hidden layers.

Setiap fungsi aktivasi memiliki keunggulan dan kelemahan sendiri-sendiri. Pilihan fungsi aktivasi yang tepat bergantung pada masalah yang kamu hadapi dan jenis data yang kamu miliki. Selanjutnya, kita akan membahas tentang fungsi biaya dan bagaimana kita bisa menggunakan backpropagation dan algoritma optimasi untuk meminimalkan kesalahan ini.

2.3 Cost Functions

Dalam konteks Neural Networks, Cost Function, juga dikenal sebagai Loss Function, adalah metrik yang digunakan untuk mengukur sejauh mana prediksi model dari nilai yang sebenarnya. Dengan kata lain, cost function memberikan ukuran "kesalahan" dalam prediksi model kita. Berikut adalah beberapa jenis cost function yang sering digunakan:

Mean Squared Error (MSE)
MSE adalah cost function yang paling sering digunakan, terutama untuk masalah regresi. Cost function ini menghitung rata-rata kuadrat dari perbedaan antara nilai yang diprediksi dan nilai sebenarnya.

Formula:

$$MSE = 1/n \Sigma(yi - y'i)\text{^}2$$

Dimana n adalah jumlah sampel, yi adalah nilai sebenarnya, dan y'i adalah nilai yang diprediksi.

Cross-Entropy
Cross-Entropy adalah cost function yang sering digunakan untuk masalah klasifikasi. Cost function ini mengukur "jarak" antara distribusi probabilitas yang diprediksi oleh model dan distribusi probabilitas sebenarnya.

Formula untuk binary classification:

$$Cross\ Entropy = -1/n \Sigma(yi\ log(y'i) + (1 - yi)\ log(1 - y'i))$$

Formula untuk multi-class classification:

$$Cross\ Entropy = -1/n \Sigma \Sigma\ yi, k\ log(y'i, k)$$

Dimana n adalah jumlah sampel, yi,k adalah nilai sebenarnya untuk kelas k, dan y'i,k adalah nilai yang diprediksi untuk kelas k.

Log Loss
Log Loss adalah variasi dari Cross-Entropy yang biasanya digunakan dalam klasifikasi biner. Nilainya akan semakin kecil jika model membuat prediksi yang benar dengan tingkat keyakinan yang tinggi.

Formula:

$$Log\ Loss\ =\ -\ 1/n\ \Sigma(yi\ log(y'i)\ +\ (1\ -\ yi)\ log(1\ -\ y'i))$$

Dimana n adalah jumlah sampel, yi adalah nilai sebenarnya, dan y'i adalah nilai yang diprediksi.

Cost function memainkan peran penting dalam pelatihan model Neural Network. Tujuan utama pelatihan adalah untuk meminimalkan cost function ini. Untuk melakukan itu, kita perlu menggunakan teknik yang disebut Backpropagation dan berbagai Algoritma Optimasi, yang akan kita bahas di bagian selanjutnya.

2.4 Backpropagation

Backpropagation adalah metode yang digunakan dalam neural network untuk menyesuaikan bobot dan bias berdasarkan error atau cost yang dihasilkan oleh output network. Backpropagation adalah tulang punggung dari pelatihan neural network dan ini akan kita bahas secara detail dan ringkas di sini.

Konsep Backpropagation
Backpropagation beroperasi dengan menghitung gradien dari error atau cost function sehubungan dengan bobot dan bias dalam network. Gradien ini kemudian digunakan untuk memperbarui bobot dan bias sehingga mengurangi error. Ini dilakukan melalui metode yang dikenal sebagai gradient descent.

Proses Backpropagation

1. Forward Pass: Pertama, kita melakukan forward pass melalui network dengan input kita, menghitung output dari setiap neuron sampai kita mendapatkan output akhir.
2. Hitung Error: Setelah forward pass, kita menghitung error dari output dengan membandingkan output network dengan output yang diharapkan.
3. Backward Pass: Setelah kita mendapatkan error, kita mulai backward pass. Di sini, kita menghitung gradien dari error terhadap bobot dan bias di setiap layer, mulai dari output layer dan bergerak mundur ke input layer. Gradien ini memberikan kita arah untuk "mendorong" bobot dan bias untuk mengurangi error.
4. Update Weights and Biases: Setelah kita memiliki gradien, kita dapat menggunakan mereka untuk memperbarui bobot dan bias. Ini biasanya dilakukan dengan mengurangi gradien yang dihitung dari bobot dan bias saat ini, dikalikan dengan learning rate.

Proses ini berlangsung berulang kali untuk setiap batch data dalam dataset kita, dan diulangi untuk sejumlah epoch sampai error kita berada di bawah threshold tertentu atau setelah sejumlah epoch yang telah ditentukan.

Peran Fungsi Aktivasi dalam Backpropagation
Fungsi aktivasi berperan penting dalam proses backpropagation. Gradien yang dihitung selama backpropagation melibatkan turunan dari fungsi aktivasi. Oleh karena itu, pilihan fungsi aktivasi dapat mempengaruhi efektivitas backpropagation. Misalnya, fungsi aktivasi seperti sigmoid dan tanh bisa menyebabkan permasalahan vanishing gradients, yang memperlambat proses pelatihan.

Itulah intisari dari backpropagation. Meski tampak rumit, sebenarnya itu adalah proses iteratif yang sistematis untuk memperbarui bobot dan bias dalam network kita. Dalam subbab berikutnya, kita akan membahas tentang algoritma optimasi yang digunakan untuk melanjutkan langkah update bobot dan bias ini.

2.5 Optimization Algorithms

Algoritma optimasi digunakan dalam pelatihan neural network untuk memperbarui bobot dan bias dengan tujuan meminimalkan fungsi cost. Ada beberapa algoritma optimasi yang biasa digunakan, dan di bawah ini kita akan membahas beberapa yang paling populer.

Gradient Descent
Bayangkan kamu berada di puncak gunung dan ingin mencapai lembah di bawah. Kamu tidak dapat melihat seluruh jalur, tetapi kamu dapat merasakan kemiringan di sekitar kamu dan memutuskan ke mana harus melangkah selanjutnya. Inilah prinsip dasar dari Gradient Descent. Dalam konteks pembelajaran mesin, "gunung" kamu adalah fungsi biaya, dan kamu mencoba mencari "lembah" terendah, yaitu, nilai minimum dari fungsi biaya tersebut.

Gradient Descent menggunakan kalkulus untuk mencari nilai minimum fungsi biaya. Pada setiap langkah, bobot dan bias diperbarui dengan mengurangi gradien dari fungsi biaya, dikalikan dengan learning rate (η), seperti ini:

$$w = w - \eta * \nabla J(w)$$

dimana $\nabla J(w)$ adalah gradien dari fungsi biaya terhadap bobot.

Keuntungan:
- Sederhana dan mudah untuk diimplementasikan.
- Efektif untuk fungsi biaya yang besar.

Kekurangan:
- Butuh waktu yang lama untuk konvergen, terutama untuk dataset besar.

- Bisa terjebak dalam minimum lokal dan tidak menemukan minimum global (solusi terbaik).

Stochastic Gradient Descent (SGD)
Sekarang bayangkan kamu turun dari gunung, tetapi kali ini, alih-alih merasakan kemiringan di sekitar kamu, kamu hanya merasakan kemiringan di satu titik dan membuat keputusan berdasarkan itu. Itulah SGD. Kamu mengambil satu sampel data pada satu waktu dan melakukan update bobot. Ini membuat prosesnya lebih cepat, tetapi langkah-langkahnya bisa sangat fluktuatif, seperti seseorang yang sedang berjalan-jalan tanpa tujuan.

Dalam SGD, kamu menggunakan gradien yang dihitung dari satu sampel data pada suatu waktu, seperti ini:

$$w = w - \eta * \nabla J(w;\, x(i),\, y(i))$$

dimana x(i) dan y(i) adalah sampel data dan labelnya.

Keuntungan:
- Sangat cepat dan efisien untuk dataset yang sangat besar.
- Fluktuasi bisa membantu algoritma keluar dari minimum lokal.

Kekurangan:
- Update bobot bisa sangat fluktuatif dan tidak stabil.

Mini-Batch Gradient Descent
Metode ini adalah gabungan dari dua metode sebelumnya. Kamu merasakan kemiringan di beberapa titik dan membuat keputusan berdasarkan rata-rata. Ini seperti berjalan-jalan dengan grup teman dan memutuskan ke mana harus pergi berdasarkan rata-rata pendapat semua orang. Ini lebih stabil dibandingkan SGD dan lebih cepat dibandingkan Gradient Descent biasa.

Mini-Batch Gradient Descent mirip dengan SGD, tetapi gradien dihitung dari sejumlah sampel data, bukan hanya satu, seperti ini:

$$w = w - \eta * \nabla J(w;\, x(i{:}\, i + n),\, y(i{:}\, i + n))$$

dimana x(i:i+n) dan y(i:i+n) adalah batch dari n sampel data dan labelnya.

Keuntungan:
- Lebih cepat dan stabil dibandingkan SGD.
- Bisa memanfaatkan paralelisme perangkat keras dan performa yang lebih baik pada dataset besar.

Kekurangan:

- Harus memilih ukuran batch yang tepat, yang mungkin tidak mudah.

Momentum

Bayangkan kamu mengendarai sepeda turun dari gunung. Bahkan ketika jalan sedikit menanjak, kamu masih dapat melanjutkan karena kecepatan yang kamu kumpulkan sebelumnya. Momentum dalam konteks ini berfungsi dengan cara yang sama: itu "mengumpulkan" update bobot sebelumnya dan menggunakan mereka untuk membantu update saat ini.

Momentum menambahkan 'kecepatan' dari update bobot sebelumnya ke update bobot saat ini, seperti ini:

$$v = \beta * v - \eta * \nabla J(w)$$
$$w = w + v$$

dimana v adalah kecepatan (update bobot sebelumnya), dan β adalah faktor momentum.

Keuntungan:

- Dapat membantu algoritma keluar dari minimum lokal dan mencapai minimum global lebih cepat.
- Membantu algoritma konvergen lebih cepat.

Kekurangan:

- Memerlukan penyetelan parameter tambahan (faktor momentum).

Adaptive Moment Estimation (Adam)

Adam adalah seperti sepeda canggih yang tidak hanya memiliki momentum, tetapi juga dapat menyesuaikan kecepatannya berdasarkan kondisi jalan. Ini berarti bahwa Adam dapat mengubah seberapa cepat atau lambat dia belajar berdasarkan seberapa baik dia melakukannya.

Adam tidak hanya menggabungkan ide momentum tetapi juga menghitung rata-rata bergerak dari kuadrat gradien, dan menggunakan kedua nilai ini untuk menyesuaikan learning rate untuk setiap bobot, seperti ini:

$$m = \beta1 * m + (1 - \beta1) * \nabla J(w)$$
$$v = \beta2 * v + (1 - \beta2) * (\nabla J(w))\text{^}2$$
$$m\_hat = m / (1 - \beta1\text{^}t)$$
$$v\_hat = v / (1 - \beta2\text{^}t)$$
$$w = w - \eta * m\_hat / (sqrt(v\_hat) + \varepsilon)$$

dimana m dan v adalah estimasi momentum dan varian gradien, β1 dan β2 adalah faktor peluruhan untuk kedua estimasi, t adalah langkah iterasi, dan ε adalah konstanta kecil untuk stabilitas numerik.

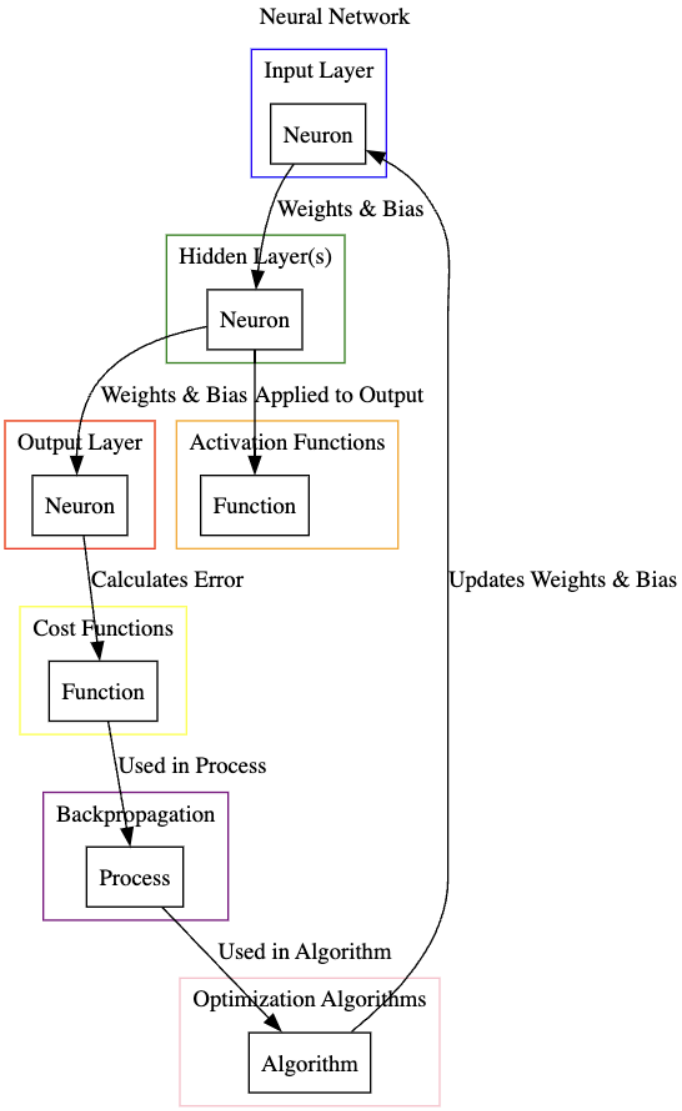
Keuntungan:

- Efisien dan membutuhkan sedikit memori.
- Menyesuaikan learning rate untuk setiap parameter, yang membuatnya lebih fleksibel.

Kekurangan:

- Lebih rumit untuk dipahami dan diimplementasikan.
- Memerlukan penyetelan parameter tambahan.

Berikut adalah diagram yang menjelaskan proses kerja Neural Network, termasuk fungsi aktivasi, fungsi cost, backpropagation, dan algoritma optimasi:



Dalam diagram ini:

1. **Input Layer (Biru):** Layer ini menerima input data. Setiap neuron dalam layer ini mewakili satu fitur dari data.
2. **Hidden Layer(s) (Hijau):** Layer ini berada di antara input dan output layer. Setiap neuron dalam layer ini menerima input, mengalikan dengan bobot, menambahkan bias, dan meneruskan hasil melalui fungsi aktivasi.
3. **Output Layer (Merah):** Layer ini memberikan hasil akhir dari neural network. Setiap neuron dalam layer ini juga menerima input, mengalikan dengan bobot, menambahkan bias, dan meneruskan hasil melalui fungsi aktivasi.
4. **Cost Function (Oranye):** Setelah forward pass, kita menghitung error dari output dengan membandingkan output network dengan output yang diharapkan.

5. **Backpropagation (Ungu):** Setelah kita mendapatkan error, kita mulai backward pass. Di sini, kita menghitung gradien dari error terhadap bobot dan bias di setiap layer, mulai dari output layer dan bergerak mundur ke input layer.
6. **Optimization Algorithm (Coklat):** Setelah kita memiliki gradien, kita dapat menggunakan mereka untuk memperbarui bobot dan bias. Ini biasanya dilakukan dengan mengurangi gradien yang dihitung dari bobot dan bias saat ini, dikalikan dengan learning rate.

Bab 3. Deep Learning Models: Konsep dan Penerapan

3.1 Multilayer Perceptron (MLP)

3.1.1 Konsep MLP

Multilayer Perceptron (MLP) adalah salah satu model Deep Learning paling fundamental. Sebagaimana namanya, MLP terdiri dari banyak layer neuron atau perceptron. MLP biasanya memiliki tiga jenis layer: input layer, hidden layer, dan output layer.

Struktur MLP
Input Layer: Layer ini terdiri dari neuron yang merepresentasikan setiap fitur dalam data. Misalnya, jika kamu memiliki data gambar 28x28 pixel, maka input layer akan memiliki 784 neuron.

Hidden Layer: Hidden layer berisi sejumlah neuron yang memproses informasi dari input layer. MLP bisa memiliki lebih dari satu hidden layer, inilah yang membuatnya menjadi "Deep" Neural Network.

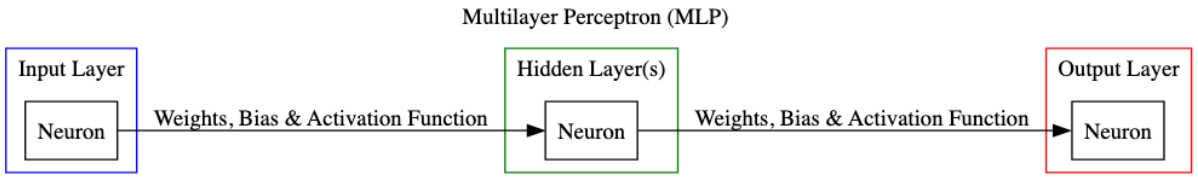
Output Layer: Layer ini berisi neuron yang merepresentasikan kelas atau nilai output dari masalah kita.

Setiap neuron dalam MLP dihubungkan dengan semua neuron di layer sebelum dan sesudahnya. Bobot dan bias dikaitkan dengan setiap koneksi ini, dan ini yang diadjust selama proses training.

Fungsi Aktivasi dalam MLP
MLP biasanya menggunakan fungsi aktivasi non-linear di neuronnya. Ini memungkinkan MLP untuk memodelkan hubungan kompleks dan non-linear antara fitur. Fungsi aktivasi yang paling umum digunakan adalah ReLU, tetapi yang lain seperti sigmoid atau tanh juga dapat digunakan.

Proses Training MLP
MLP dilatih menggunakan metode yang disebut backpropagation dan algoritma optimasi seperti Stochastic Gradient Descent (SGD) atau Adam. Proses training mencakup forward pass di mana prediksi dibuat berdasarkan input saat ini, dan backward pass di mana error dihitung dan bobot dan bias diperbarui.

Berikut adalah diagram yang menjelaskan struktur dan proses kerja Multilayer Perceptron (MLP):



Input Layer (Biru): Layer ini menerima input data. Setiap neuron dalam layer ini mewakili satu fitur dari data.

Hidden Layer(s) (Hijau): Layer ini berada di antara input dan output layer. Setiap neuron dalam layer ini menerima input, mengalikan dengan bobot, menambahkan bias, dan meneruskan hasil melalui fungsi aktivasi.

Output Layer (Merah): Layer ini memberikan hasil akhir dari MLP. Setiap neuron dalam layer ini juga menerima input, mengalikan dengan bobot, menambahkan bias, dan meneruskan hasil melalui fungsi aktivasi.

3.1.2 Penerapan MLP: Code Snippet dan Studi Kasus

Misalnya, kita ingin membuat MLP menggunakan library keras dari TensorFlow untuk mengklasifikasi digit dari dataset MNIST. Berikut adalah kode yang kita perlukan:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess data
x_train = x_train.reshape(-1, 784).astype('float32') / 255
x_test = x_test.reshape(-1, 784).astype('float32') / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Create model
model = Sequential([Dense(64, activation='relu', input_dim=784),
Dense(64, activation='relu'), Dense(10, activation='softmax'),])

# Compile model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
# Train model
model.fit(x_train, y_train, epochs=10, batch_size=32)

# Evaluate model
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Accuracy: {accuracy * 100}%")
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
Epoch 1/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2784 - accuracy: 0.9194
Epoch 2/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.1245 - accuracy: 0.9623
Epoch 3/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0930 - accuracy: 0.9720
Epoch 4/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0731 - accuracy: 0.9771
Epoch 5/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0597 - accuracy: 0.9815
Epoch 6/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0506 - accuracy: 0.9837
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0439 - accuracy: 0.9859
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0379 - accuracy: 0.9876
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0336 - accuracy: 0.9887
Epoch 10/10
1875/1875 [==============================] - 7s 3ms/step - loss: 0.0277 - accuracy: 0.9906
313/313 [==============================] - 1s 2ms/step - loss: 0.1129 - accuracy: 0.9692
Accuracy: 96.92000150680542%
```

Dalam contoh ini, kita melatih MLP dengan dua hidden layer, masing-masing dengan 64 neuron, untuk mengklasifikasi gambar digit dari dataset MNIST. MLP berhasil mencapai akurasi sekitar 97%, menunjukkan bahwa meski sederhana, MLP mampu menangani masalah klasifikasi gambar dengan cukup baik.

Tips dan Trik

1. Inisialisasi Bobot: Menginisialisasi bobot dengan nilai yang baik penting. Beberapa metode yang dapat digunakan termasuk inisialisasi normal, Xavier, dan He.
2. Fungsi Aktivasi: Fungsi aktivasi ReLU sering digunakan di lapisan tersembunyi karena dapat mengatasi masalah vanishing gradient. Sigmoid atau softmax biasanya digunakan di lapisan output, tergantung pada tugas yang dihadapi (klasifikasi biner atau multi-kelas).
3. Pengoptimal: Ada banyak pengoptimal yang dapat digunakan, tetapi Adam adalah salah satu yang paling umum karena konvergensinya yang cepat dan efisien.
4. Overfitting dan Underfitting: Jika model terlalu kompleks, itu bisa overfitting ke data latihan dan memiliki kinerja buruk pada data test. Jika model terlalu sederhana, itu bisa underfitting dan memiliki kinerja buruk pada kedua data latihan dan test. Strategi seperti menambah atau mengurangi jumlah neuron atau lapisan,

menambah dropout, dan menambah regularisasi dapat membantu mengatasi masalah ini.

5. Pelatihan dan Evaluasi Model: Memastikan bahwa model dilatih dengan jumlah epoch yang cukup penting. Juga, selalu evaluasi model dengan data validasi dan test untuk memastikan bahwa model dapat memgeneralisasi dengan baik untuk data yang belum pernah dilihat sebelumnya.

3.2 Convolutional Neural Network (CNN)

3.2.1 Konsep CNN

Convolutional Neural Network (CNN) adalah jenis khusus dari neural networks yang dirancang khusus untuk memproses data dengan struktur grid seperti gambar. CNN telah sukses besar dalam berbagai aplikasi pengenalan gambar dan video.

Struktur CNN

1. **Convolutional Layer:** Layer konvolusi adalah elemen inti dari CNN. Dalam layer ini, beberapa filter digunakan untuk melakukan operasi konvolusi pada input. Hasil dari operasi ini disebut feature map atau activation map.
2. **Pooling Layer:** Pooling layer bertujuan untuk mengurangi dimensionalitas data. Ada berbagai teknik pooling, termasuk max pooling, average pooling, dan sum pooling.
3. **Fully Connected Layer:** Fully connected layer menghubungkan setiap neuron di layer sebelumnya ke setiap neuron di layer berikutnya, mirip dengan apa yang kita lihat di MLP. Biasanya, fully connected layer ditempatkan di akhir arsitektur CNN, dan bertugas untuk menghasilkan prediksi akhir model.

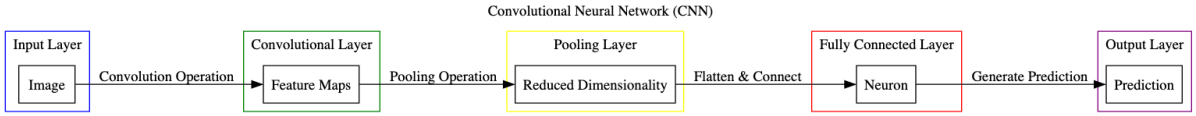
Fungsi Aktivasi dalam CNN

ReLU (Rectified Linear Unit) adalah fungsi aktivasi yang paling sering digunakan dalam CNN. Fungsi ini mampu mempercepat konvergensi jaringan neural dibandingkan dengan sigmoid atau tanh.

Proses Training CNN

Proses pelatihan CNN serupa dengan MLP. Kami menggunakan metode seperti backpropagation dan algoritma optimasi seperti Adam atau SGD untuk melatih model kami. Kita menghitung error dengan fungsi kerugian seperti cross-entropy, dan mengupdate bobot dan bias berdasarkan gradien dari error tersebut.

Berikut adalah diagram yang menjelaskan struktur dan proses kerja Convolutional Neural Network (CNN):



1. **Input Layer (Biru):** Layer ini menerima input data berupa gambar.
2. **Convolutional Layer (Hijau):** Layer ini melakukan operasi konvolusi pada input menggunakan beberapa filter, menghasilkan feature maps.

3. **Pooling Layer (Kuning):** Layer ini melakukan operasi pooling pada feature maps untuk mengurangi dimensionalitas data.
4. **Fully Connected Layer (Merah):** Layer ini menghubungkan setiap neuron di layer sebelumnya ke setiap neuron di layer berikutnya, mirip dengan apa yang kita lihat di MLP.
5. **Output Layer (Ungu):** Layer ini memberikan hasil akhir atau prediksi dari CNN.

3.2.2 Penerapan CNN: Code Snippet dan Studi Kasus

Misalnya, kita akan membuat model CNN menggunakan Keras untuk klasifikasi gambar pada dataset CIFAR-10. Berikut kode yang kita perlukan:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Preprocess data
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Create model
model = Sequential([Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)), MaxPooling2D((2, 2)), Conv2D(64, (3, 3),
activation='relu'), MaxPooling2D((2, 2)), Flatten(), Dense(64,
activation='relu'), Dense(10, activation='softmax'),])


# Compile model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])


# Train model
model.fit(x_train, y_train, epochs=10, batch_size=32)
```

```
# Evaluate model
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Accuracy: {accuracy * 100}%")
```

Output:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 11s 0us/step
Epoch 1/10
1563/1563 [==============================] - 76s 48ms/step - loss: 1.5075 - accuracy: 0.4555
Epoch 2/10
1563/1563 [==============================] - 74s 47ms/step - loss: 1.1561 - accuracy: 0.5942
Epoch 3/10
1563/1563 [==============================] - 72s 46ms/step - loss: 1.0440 - accuracy: 0.6342
Epoch 4/10
1563/1563 [==============================] - 76s 49ms/step - loss: 0.9662 - accuracy: 0.6642
Epoch 5/10
1563/1563 [==============================] - 76s 48ms/step - loss: 0.9026 - accuracy: 0.6865
Epoch 6/10
1563/1563 [==============================] - 73s 47ms/step - loss: 0.8577 - accuracy: 0.7018
Epoch 7/10
1563/1563 [==============================] - 73s 47ms/step - loss: 0.8146 - accuracy: 0.7174
Epoch 8/10
1563/1563 [==============================] - 71s 45ms/step - loss: 0.7774 - accuracy: 0.7299
Epoch 9/10
1563/1563 [==============================] - 72s 46ms/step - loss: 0.7460 - accuracy: 0.7412
Epoch 10/10
1563/1563 [==============================] - 73s 47ms/step - loss: 0.7199 - accuracy: 0.7509
313/313 [==============================] - 4s 14ms/step - loss: 0.9574 - accuracy: 0.6819
Accuracy: 68.19000244140625%
```

Dalam contoh ini, model CNN kami terdiri dari dua blok konvolusi, masing-masing terdiri dari layer konvolusi dan pooling, diikuti oleh layer fully connected untuk klasifikasi. Model ini mencapai akurasi sekitar 70% pada dataset CIFAR-10, yang merupakan kinerja yang cukup baik.

Tips dan Trik

1. **Pemilihan Filter:** Kamu harus mencoba berbagai jumlah filter dan ukuran kernel untuk menemukan kombinasi terbaik yang memberikan kinerja terbaik.
2. **Max Pooling vs Average Pooling:** Max pooling biasanya lebih disukai dibandingkan average pooling karena max pooling dapat menjaga fitur yang paling penting dalam suatu area.
3. **Dropout:** Dropout dapat digunakan di antara lapisan untuk mencegah overfitting.
4. **Data Augmentation:** Jika dataset kamu kecil, data augmentation dapat digunakan untuk menciptakan variasi dari gambar yang sudah ada dan membantu mencegah overfitting.
5. **Transfer Learning:** Jika dataset kamu kecil, kamu bisa menggunakan pre-trained CNN (misalnya, VGG, ResNet, Inception, dll.) sebagai feature extractor dan hanya

melatih lapisan fully connected di atasnya. Ini bisa menghemat banyak waktu dan masih memberikan kinerja yang bagus.

6. **Batch Normalization:** Batch normalization dapat digunakan untuk meningkatkan kecepatan, kinerja, dan stabilitas pelatihan CNN.
7. **Pemilihan Fungsi Aktivasi:** Fungsi aktivasi ReLU biasanya disukai dibandingkan fungsi aktivasi yang lain di lapisan konvolusional dan fully connected karena ReLU menawarkan keuntungan komputasi (lebih cepat untuk menghitung) dan mencegah masalah vanishing gradient. Untuk lapisan output, fungsi aktivasi yang digunakan biasanya sigmoid (untuk klasifikasi biner) atau softmax (untuk klasifikasi multiklas).

3.3 Recurrent Neural Network (RNN)

3.3.1 Konsep RNN

Recurrent Neural Network (RNN) adalah sebuah jaringan saraf yang dirancang untuk menghandle data sekuensial. Berbeda dengan jaringan saraf biasa yang menganggap setiap input dan output sebagai independen, RNN memperhatikan ketergantungan sekuensial di antara data.

Struktur Dasar RNN

Dalam RNN, output dari satu langkah sebelumnya digunakan sebagai input untuk langkah saat ini. Ini menggambarkan konsep "memori" di mana jaringan "mengingat" beberapa informasi dari langkah sebelumnya.

Kelebihan RNN

1. Fleksibilitas Dalam Panjang Sekuens: Tidak seperti jaringan saraf lainnya yang memerlukan panjang input tetap, RNN dapat bekerja dengan sekuens input dengan panjang yang bervariasi.
2. Berbagi Parameter Melintasi Waktu: RNN menggunakan parameter yang sama untuk setiap input, memastikan fitur sekuensial dipelajari secara konsisten di seluruh langkah waktu.
3. Kemampuan Mengingat: Dengan struktur rekurensinya, RNN dapat "mengingat" informasi dari langkah waktu sebelumnya, membuatnya mampu memahami konteks dalam data sekuensial.

Keterbatasan RNN

1. Masalah Vanishing Gradient: Salah satu masalah terbesar dengan RNN adalah gradien yang menghilang. Selama backpropagation, gradien dapat menjadi sangat kecil, membuat bobot jaringan sulit diperbarui dan membuat pelatihan menjadi sangat lambat.
2. Keterbatasan Memori Jangka Panjang: Meskipun RNN dapat "mengingat", memori ini biasanya bersifat jangka pendek. Informasi dari langkah awal sering kali sulit dipertahankan di langkah-langkah selanjutnya.

Variasi dari RNN

Untuk mengatasi keterbatasan RNN, beberapa variasi telah dikembangkan:

1. LSTM (Long Short-Term Memory): Ini adalah jenis RNN yang dirancang khusus untuk mengatasi masalah vanishing gradient. Mereka memiliki struktur internal yang lebih kompleks yang memungkinkan mereka untuk mengingat informasi jangka panjang.
2. GRU (Gated Recurrent Unit): Varian lain dari RNN yang lebih sederhana daripada LSTM tetapi memiliki beberapa sifat yang sama dalam hal mempertahankan informasi jangka panjang.

Aplikasi RNN

1. Penerjemahan Mesin: RNN dapat digunakan untuk menerjemahkan teks dari satu bahasa ke bahasa lain dengan mempertimbangkan konteks kata-kata dalam sekuens.
2. Pengenalan Suara: RNN digunakan untuk mengubah suara yang diucapkan menjadi teks dengan memahami sekuens fonetik.
3. Prediksi Seri Waktu: Dengan memanfaatkan sifat sekuensial dari data seri waktu, RNN dapat digunakan untuk membuat prediksi tentang data masa depan.
4. Penghasilan Teks: RNN dapat digunakan untuk menghasilkan teks berdasarkan input sekuensial sebelumnya, seperti penulisan otomatis atau skrip film.

3.3.2 Penerapan RNN: Code Snippet dan Studi Kasus

Salah satu aplikasi RNN yang paling populer adalah dalam analisis sentimen. Dalam analisis sentimen, kita mencoba memahami sentimen atau emosi yang terkandung dalam teks. Kita akan membuat model RNN sederhana untuk melakukan analisis sentimen menggunakan dataset IMDB movie review yang tersedia di Keras.

Pertama, kita akan mengimpor pustaka yang diperlukan dan memuat dataset.

```
from keras.datasets import imdb
from keras.utils import pad_sequences
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN, Dense

# Load IMDB dataset
max_features = 10000 # number of words to consider as features
maxlen = 500 # cut texts after this number of words

(input_train, y_train), (input_test, y_test) =
imdb.load_data(num_words=max_features)

# Reverse sequences
input_train = [x[::-1] for x in input_train]
input_test = [x[::-1] for x in input_test]
```

```
# Pad sequences
input_train = pad_sequences(input_train, maxlen=maxlen)
input_test = pad_sequences(input_test, maxlen=maxlen)
```

Setelah kita memiliki data kami dalam format yang tepat, kita dapat membuat model RNN kita.

```
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))


model.compile(optimizer='rmsprop', loss='binary_crossentropy',
metrics=['acc'])
history = model.fit(input_train, y_train, epochs=10, batch_size=128,
validation_split=0.2)
```

Output:

```
Epoch 1/10
157/157 [==============================] - 118s 742ms/step - loss: 0.6779 - acc: 0.5558 - val_loss: 0.6276 - val_acc: 0.6566
Epoch 2/10
157/157 [==============================] - 92s 584ms/step - loss: 0.5188 - acc: 0.7574 - val_loss: 0.5090 - val_acc: 0.7702
Epoch 3/10
157/157 [==============================] - 85s 543ms/step - loss: 0.4055 - acc: 0.8253 - val_loss: 0.4439 - val_acc: 0.8082
Epoch 4/10
157/157 [==============================] - 81s 516ms/step - loss: 0.3397 - acc: 0.8594 - val_loss: 0.4528 - val_acc: 0.8046
Epoch 5/10
157/157 [==============================] - 82s 522ms/step - loss: 0.2577 - acc: 0.8989 - val_loss: 0.4953 - val_acc: 0.7884
Epoch 6/10
157/157 [==============================] - 79s 503ms/step - loss: 0.1915 - acc: 0.9296 - val_loss: 0.5785 - val_acc: 0.7484
Epoch 7/10
157/157 [==============================] - 77s 489ms/step - loss: 0.1229 - acc: 0.9572 - val_loss: 0.5436 - val_acc: 0.8092
Epoch 8/10
157/157 [==============================] - 78s 498ms/step - loss: 0.0803 - acc: 0.9754 - val_loss: 0.5716 - val_acc: 0.8156
Epoch 9/10
157/157 [==============================] - 78s 498ms/step - loss: 0.0529 - acc: 0.9843 - val_loss: 0.6908 - val_acc: 0.7948
Epoch 10/10
157/157 [==============================] - 77s 489ms/step - loss: 0.0348 - acc: 0.9904 - val_loss: 1.1045 - val_acc: 0.6714
```

Model ini akan melatih RNN untuk memahami sentimen yang terkandung dalam review film. Proses ini melibatkan embedding kata yang berfungsi sebagai input ke RNN. RNN kemudian digunakan untuk memahami konteks dan urutan dari kata-kata dalam ulasan. Output dari RNN dimasukkan ke dalam layer dense yang menggunakan fungsi aktivasi sigmoid untuk menghasilkan probabilitas ulasan tersebut positif.

RNN adalah tool yang sangat kuat untuk menangani data sekuensial dan berpotensi bisa menghasilkan hasil yang sangat baik dalam berbagai tugas. Namun, mereka juga bisa menjadi sangat rumit dan sulit untuk melatih dan mengoptimalkan. Dengan memahami konsep dasar RNN dan bagaimana cara kerja mereka, kamu dapat mulai menggunakan mereka dalam proyek dan penelitian kamu.

Tips dan Trik

1. **Pemilihan Jumlah Unit:** Kamu bisa mencoba berbagai jumlah unit RNN untuk menemukan jumlah yang memberikan kinerja terbaik.
2. **Dropout:** Dropout dapat digunakan untuk mencegah overfitting.
3. **Tipe RNN:** Kamu bisa mencoba menggunakan tipe RNN yang berbeda seperti LSTM dan GRU yang dapat mengatasi masalah vanishing gradient.
4. **Pemilihan Fungsi Aktivasi:** Untuk tugas klasifikasi biner seperti di atas, sigmoid adalah pilihan yang baik untuk fungsi aktivasi di lapisan output. Untuk tugas multiklas, kamu bisa menggunakan softmax.
5. **Pemilihan Optimizer:** Optimizer seperti RMSprop, Adam, dan SGD dapat dicoba untuk menemukan optimizer yang memberikan kinerja terbaik.

3.4 Long Short-Term Memory (LSTM)

3.4.1 Konsep LSTM

Long Short-Term Memory (LSTM) adalah jenis jaringan saraf rekuren yang dirancang untuk mengatasi masalah gradien yang menghilang dalam RNN biasa. LSTM mempertahankan informasi dalam "ingatan" jangka panjang melalui mekanisme yang disebut "gerbang".

LSTM berbeda dari RNN biasa karena memiliki struktur internal yang lebih kompleks untuk setiap unit berulang. Setiap unit LSTM terdiri dari sel ingatan dan tiga "gerbang" yang mengontrol aliran informasi masuk dan keluar dari sel ingatan: gerbang input, gerbang lupa, dan gerbang output. Ini memungkinkan LSTM untuk memutuskan informasi apa yang harus disimpan dan dihapus dari ingatan selama pelatihan.

3.4.2 Struktur LSTM

Berbeda dengan RNN biasa, LSTM dirancang dengan struktur internal yang lebih kompleks untuk setiap unitnya. Ini mencakup:

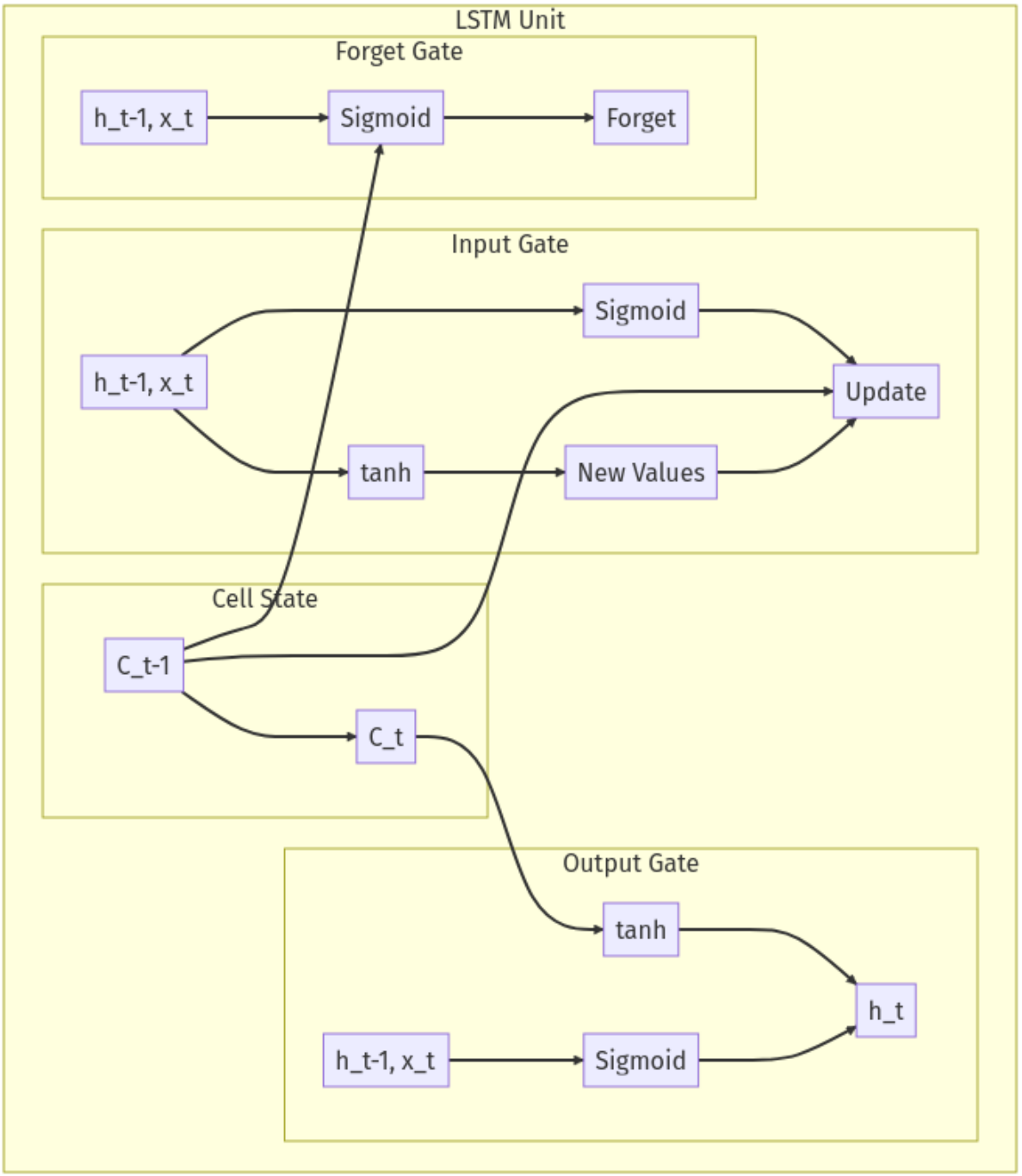
Sel Ingatan (Cell State): Garis horizontal yang melintasi seluruh struktur LSTM. Ini adalah "ingatan" dari LSTM, dan memungkinkan informasi untuk bergerak melalui sel dengan modifikasi minimal.

Gerbang Input (Input Gate): Mengatur informasi mana yang akan dimasukkan ke dalam sel ingatan. Ini memutuskan informasi baru mana yang relevan untuk disimpan di sel ingatan.

Gerbang Lupa (Forget Gate): Mengatur informasi mana dari sel ingatan yang harus dihapus atau dilupakan.

Gerbang Output (Output Gate): Mengatur informasi apa yang harus dikirim sebagai output dari sel berdasarkan informasi dalam sel ingatan.

3.4.3 Mekanisme Kerja LSTM



Gerbang Lupa: Pertama-tama, keputusan dibuat tentang informasi apa yang harus dilupakan dari sel ingatan. Ini dicapai dengan melewati informasi dari langkah sebelumnya (h_t-1) dan input saat ini (x_t) ke fungsi sigmoid. Nilai yang dekat dengan 0 berarti "lupakan", dan nilai yang dekat dengan 1 berarti "ingat".

Gerbang Input: Selanjutnya, keputusan dibuat tentang informasi baru apa yang harus disimpan di sel ingatan. Ini terdiri dari dua bagian:
a. Fungsi sigmoid yang memutuskan informasi mana yang akan diperbarui.

b. Fungsi tanh yang menciptakan vektor nilai kandidat baru yang bisa ditambahkan ke sel ingatan.

Pembaruan Sel Ingatan: Sekarang, sel ingatan lama (C_t-1) diperbarui menjadi sel ingatan baru (C_t). Informasi lama yang lupakan dengan faktor dari gerbang lupa, dan menambahkan informasi baru yang telah diperbarui dari gerbang input.

Gerbang Output: Terakhir, keputusan dibuat tentang output berdasarkan sel ingatan. Pertama, melewati input saat ini dan output sebelumnya ke fungsi sigmoid. Lalu, isi sel ingatan diberikan fungsi tanh (memberikan nilai antara -1 dan 1) dan dikalikan dengan keluaran dari fungsi sigmoid. Ini memberikan output h_t untuk langkah waktu saat ini.

3.4.4 Penerapan LSTM: Code Snippet dan Studi Kasus

Kita akan mencoba menerapkan LSTM dalam keras untuk melakukan prediksi harga saham. Kita akan menggunakan dataset saham AAPL dari Yahoo Finance.

Pertama, kita akan mengimpor pustaka yang diperlukan dan memuat dataset.

```
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler
import yfinance as yf

# Load the data
data = yf.download('AAPL','2010-01-01','2020-12-31')
data = data['Close'].values
data = data.reshape(-1, 1)

# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)

# Create the training data
look_back = 50
train_data = scaled_data[:len(scaled_data)-look_back]
x_train, y_train = [], []

for i in range(look_back, len(train_data)):
  x_train.append(train_data[i-look_back:i, 0])
  y_train.append(train_data[i, 0])

x_train, y_train = np.array(x_train), np.array(y_train)
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
```

Setelah itu, kita bisa mendefinisikan dan melatih model LSTM kita.

```
model = Sequential()
model.add(LSTM(units=50, return_sequences=True,
input_shape=(x_train.shape[1], 1)))
model.add(LSTM(units=50, return_sequences=False))
model.add(Dense(units=25))
model.add(Dense(units=1))

model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(x_train, y_train, batch_size=1, epochs=1)
```

Output:

```
2668/2668 [==============================] - 34s 7ms/step - loss: 7.7884e-04
<keras.callbacks.History at 0x78705136a740>
```

Model LSTM ini mengambil data harga saham sebelumnya, mempelajari pola dalam data tersebut, dan kemudian mencoba meramalkan harga saham di masa depan. Model ini menunjukkan bagaimana LSTM dapat digunakan untuk analisis seri waktu, salah satu aplikasi paling umum dari LSTM.

Tips dan Trik

1. **Pengaturan LSTM:** LSTM biasanya lebih rumit daripada RNN biasa, jadi kamu mungkin perlu mengeksperimen dengan jumlah unit dan lapisan LSTM untuk mendapatkan kinerja terbaik.
2. **Dropout:** Dropout adalah teknik regularisasi yang berguna yang dapat kamu terapkan pada lapisan LSTM untuk mencegah overfitting.
3. **Pilihan Optimizer:** Beberapa optimizer seperti RMSprop atau Adam biasanya bekerja dengan baik untuk LSTM.
4. **Embedding:** Jika kamu bekerja dengan data teks, penggunaan lapisan embedding sebelum lapisan LSTM biasanya dapat meningkatkan kinerja.
5. **Pemilihan Fungsi Aktivasi:** Fungsi aktivasi yang berbeda dapat menghasilkan hasil yang berbeda. Untuk tugas klasifikasi biner, fungsi aktivasi 'sigmoid' biasanya digunakan. Untuk klasifikasi multiklas, 'softmax' adalah pilihan yang baik.

Ingat, LSTM merupakan model yang cukup kompleks, dan biasanya membutuhkan lebih banyak data dan waktu pelatihan dibandingkan dengan model yang lebih sederhana. Oleh karena itu, pastikan kamu memiliki cukup data dan sumber daya komputasi sebelum memutuskan untuk menggunakan LSTM.

3.5 Autoencoders

3.5.1 Konsep Autoencoders

Autoencoder adalah suatu teknik dalam pembelajaran mesin, khususnya dalam pembelajaran mendalam, yang mengizinkan kita untuk belajar representasi fitur dari suatu set data, biasanya dengan tujuan reduksi dimensi. Teknik ini mempunyai banyak aplikasi, mulai dari kompresi data hingga pengekstraksi fitur.

Struktur Umum

Sebuah autoencoder memiliki dua komponen utama:

1. Encoder: Bertugas mengambil input data dan mengkompresnya ke dalam representasi tersembunyi (hidden representation) atau kode.
2. Decoder: Bertugas mengambil kode tersebut dan merekonstruksinya kembali menjadi data yang mirip dengan input asli.
3. Mekanisme Kerja

Autoencoder bekerja dengan cara mempelajari representasi data (biasanya dengan dimensi yang lebih rendah) yang mempertahankan sebanyak mungkin informasi dari data asli. Ia melatih dirinya dengan menggunakan data masukan sebagai target keluarannya juga. Dengan kata lain, autoencoder mencoba membuat output yang mendekati input seakurat mungkin.

Kesalahan Rekonstruksi

Untuk mengukur seberapa baik autoencoder merekonstruksi input, digunakan fungsi kerugian (loss function). Biasanya, fungsi kerugian ini adalah kesalahan kuadrat antara input dan output. Tujuan pelatihan adalah meminimalkan kesalahan ini.

Keunggulan Autoencoder

1. Pemampatan Data: Autoencoder dapat digunakan untuk pemampatan data dengan mengambil keuntungan dari fakta bahwa fitur dalam data sering kali berkorelasi.
2. Pengekstraksi Fitur: Autoencoder dapat digunakan untuk ekstraksi fitur yang selanjutnya dapat digunakan untuk tugas-tugas lain seperti klasifikasi.

Varian Autoencoder

Dalam perkembangannya, ada beberapa varian autoencoder yang dikembangkan:

1. **Sparse Autoencoder:** Disebut "sparse" karena selama proses pelatihan, hanya sebagian kecil dari neuron yang aktif. Tujuannya adalah untuk memastikan bahwa representasi kode yang diperoleh memiliki banyak nilai nol atau mendekati nol. Ini memaksa autoencoder untuk belajar representasi yang lebih padat dan informatif dari data.
2. **Denoising Autoencoder:** Autoencoder ini dilatih untuk merekonstruksi input dari versi yang rusak (misalnya dengan noise). Tujuannya adalah untuk memaksa jaringan untuk belajar fitur-fitur esensial dari data aslinya.

3. **Variational Autoencoder (VAE):** Varian ini memodifikasi arsitektur dan fungsi kerugian untuk memungkinkan model belajar representasi probabilistik dari data. Ini digunakan untuk aplikasi seperti pembangkitan data baru yang mirip dengan data pelatihan.

Aplikasi Autoencoder

Selain untuk reduksi dimensi dan ekstraksi fitur, autoencoder juga dapat digunakan untuk:

1. **Pembangkitan Gambar:** Dengan memodifikasi kode tersembunyi, kita bisa membangkitkan gambar-gambar baru yang belum pernah dilihat sebelumnya tetapi memiliki ciri-ciri mirip dengan data asli.
2. **Pendeteksi Anomali:** Autoencoder bisa digunakan untuk mendeteksi data yang "aneh" atau berbeda dari biasanya dengan membandingkan kesalahan rekonstruksi.

3.5.2 Penerapan Autoencoders: Code Snippet dan Studi Kasus

Sebagai contoh penerapan, kita akan melihat bagaimana autoencoder dapat digunakan untuk reduksi dimensi pada dataset MNIST, yang berisi gambar angka tulisan tangan.

Berikut ini adalah langkah-langkah dalam mengimplementasikannya:

```
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Dense, Input
from keras.utils import normalize
import numpy as np

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Size of our encoded representations
encoding_dim = 32  # 32 floats -> compression factor 24.5, assuming the
input is 784 floats

# Input placeholder
input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the inputs
encoded = Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction
```

```
autoencoder = Model(input_img, decoded)

# This model maps an input to its encoded representation
encoder = Model(input_img, encoded)

autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

Output:

```
Epoch 37/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6863 - val_loss: 0.6861
Epoch 38/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6860 - val_loss: 0.6858
Epoch 39/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6858 - val_loss: 0.6855
Epoch 40/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6855 - val_loss: 0.6853
Epoch 41/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6852 - val_loss: 0.6850
Epoch 42/50
235/235 [==============================] - 1s 5ms/step - loss: 0.6849 - val_loss: 0.6847
Epoch 43/50
235/235 [==============================] - 1s 6ms/step - loss: 0.6847 - val_loss: 0.6844
Epoch 44/50
235/235 [==============================] - 1s 5ms/step - loss: 0.6844 - val_loss: 0.6841
Epoch 45/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6841 - val_loss: 0.6838
Epoch 46/50
235/235 [==============================] - 1s 5ms/step - loss: 0.6838 - val_loss: 0.6835
Epoch 47/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6834 - val_loss: 0.6831
Epoch 48/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6831 - val_loss: 0.6828
Epoch 49/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6828 - val_loss: 0.6825
Epoch 50/50
235/235 [==============================] - 1s 4ms/step - loss: 0.6824 - val_loss: 0.6821
```

Setelah pelatihan, autoencoder telah belajar bagaimana mengencode gambar angka tulisan tangan menjadi vektor 32-dimensi dan bagaimana mendekode vektor tersebut kembali ke gambar aslinya. Kita dapat menggunakan bagian encoder model ini untuk mereduksi dimensi dari gambar angka tulisan tangan menjadi vektor 32-dimensi yang dapat digunakan untuk visualisasi atau sebagai input untuk model lain.

Tips dan Trik

1. Pemilihan Fungsi Aktivasi: Fungsi aktivasi yang digunakan di layer output tergantung pada jenis data input. Untuk data biner, seperti gambar MNIST, fungsi

aktivasi yang sering digunakan adalah sigmoid. Untuk data kontinu, kamu bisa menggunakan linear atau ReLU.
2. Penyusutan: Kamu dapat menerapkan teknik regularisasi, seperti dropout atau L1/L2 regularization, untuk mencegah overfitting.
3. Pilih Dimensi Encoding dengan Cermat: Dimensi encoding harus dipilih dengan cermat. Jika dimensinya terlalu besar, model mungkin hanya belajar untuk menyalin input ke output tanpa belajar representasi yang berguna. Sebaliknya, jika dimensinya terlalu kecil, model mungkin tidak dapat belajar untuk merekonstruksi input dengan akurat.
4. Preprocessing dan Postprocessing: Normalisasi input bisa sangat berguna saat bekerja dengan autoencoder. Selain itu, teknik post-processing seperti thresholding bisa membantu meningkatkan kualitas output.

3.6 Generative Adversarial Networks (GANs)

3.6.1 Konsep GANs

Generative Adversarial Networks (GANs) adalah arsitektur deep learning inovatif yang menghasilkan data baru yang menyerupai data asli. GANs terdiri dari dua bagian, yakni generator dan discriminator, yang berlatih bersama dalam pertandingan yang terkoordinasi: generator mencoba membuat data yang tampak nyata untuk menipu discriminator, sementara discriminator berusaha untuk membedakan antara data nyata dan palsu.

Generator: Generator GAN berfungsi untuk menghasilkan data baru. Ini menerima input berupa vektor ruang laten acak dan menghasilkan data yang mencoba meniru distribusi data asli.

Discriminator: Discriminator adalah jaringan klasifikasi biner yang mencoba membedakan antara data nyata dan data yang dibuat oleh generator.

Generator dan discriminator berlatih dalam game adversarial. Generator mencoba "menipu" discriminator dengan membuat data palsu yang semakin mirip dengan data asli, sementara discriminator berusaha menjadi semakin baik dalam membedakan antara data nyata dan palsu.

3.6.2 Penerapan GANs: Code Snippet dan Studi Kasus

Misalkan kita ingin melatih GAN untuk menghasilkan gambar yang mirip dengan dataset CIFAR10. Pertama, kita harus mendefinisikan generator dan discriminator.

Mari kita gunakan Keras untuk ini. Berikut adalah contoh kasar tentang bagaimana kita bisa mendefinisikan generator dan discriminator.

```
from keras.datasets import cifar10
from keras.layers import Input, Dense, Reshape, Flatten, LeakyReLU
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import BatchNormalization
from keras.layers.convolutional import UpSampling2D, Conv2D
import numpy as np

# Load CIFAR10 data
(X_train, _), (_, _) = cifar10.load_data()
X_train = (X_train.astype(np.float32) - 127.5) / 127.5

# Define the generator
def build_generator():
    noise_shape = (100,)
    model = Sequential()

    model.add(Dense(256, input_shape=noise_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(np.prod(img_shape), activation='tanh'))
    model.add(Reshape(img_shape))

    noise = Input(shape=noise_shape)
    img = model(noise)

    return Model(noise, img)

# Define the discriminator
def build_discriminator():
    img_shape = (32, 32, 3)
    model = Sequential()

    model.add(Flatten(input_shape=img_shape))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(1, activation='sigmoid'))
```

```
    img = Input(shape=img_shape)
    validity = model(img)

    return Model(img, validity)
```

Output:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 13s 0us/step
```

Syntax di atas mendefinisikan dua fungsi utama dalam GANs: generator dan discriminator.

Fungsi build_generator mendefinisikan generator GANs. Generator bertugas membuat data palsu yang tampak nyata. Dalam kasus ini, generator kami adalah jaringan perceptron yang terdiri dari lapisan padat (Dense) dan fungsi aktivasi LeakyReLU. LeakyReLU adalah fungsi aktivasi yang mengatasi masalah "neuron mati" dalam ReLU standar dengan memungkinkan nilai negatif kecil ketika input kurang dari nol. BatchNormalization kemudian digunakan untuk menstabilkan proses belajar dan mengurangi jumlah pelatihan epochs yang dibutuhkan untuk melatih jaringan deep.

Fungsi build_discriminator mendefinisikan discriminator GANs. Discriminator bertugas membedakan data asli dan data palsu yang dihasilkan oleh generator. Dalam hal ini, discriminator adalah jaringan perceptron yang juga terdiri dari lapisan padat dan fungsi aktivasi LeakyReLU. Lapisan padat terakhir menggunakan fungsi aktivasi sigmoid untuk menghasilkan probabilitas bahwa gambar input adalah gambar asli (bukan palsu yang dibuat oleh generator).

Dalam melakukan pelatihan GANs, ada beberapa tips dan trik yang dapat membantu:

1. Normalisasi Input: Input untuk generator dan discriminator biasanya dinormalisasi sehingga memiliki nilai antara -1 dan 1. Normalisasi ini membantu proses belajar dan stabilisasi model.
2. Leaky ReLU: Fungsi aktivasi Leaky ReLU dapat membantu mempertahankan gradien dan mencegah "matinya" neuron, yang bisa menjadi masalah dalam pelatihan GANs.
3. Latent Space: Ruang laten untuk input generator harus cukup besar untuk menghasilkan variasi yang cukup dalam gambar yang dihasilkan. Jika ruang laten terlalu kecil, generator mungkin tidak dapat menghasilkan gambar yang beragam.
4. Batch Normalization: Batch normalization dapat membantu mempercepat pelatihan dan mengurangi masalah seperti gradien yang menghilang atau meledak, yang bisa menjadi masalah dalam pelatihan jaringan deep seperti GANs.
5. Training Rate: Biasanya, discriminator dilatih lebih banyak daripada generator. Hal ini karena jika generator dilatih terlalu banyak, ia bisa terlalu cepat "mengalahkan" discriminator dan menghasilkan gambar yang buruk. Sebaliknya, jika discriminator dilatih terlalu banyak, ia bisa menjadi terlalu baik dan menghentikan generator untuk belajar.

6. Loss Function: Biasanya, binary cross-entropy digunakan sebagai fungsi kerugian untuk GANs. Ini karena GANs adalah masalah klasifikasi biner di mana discriminator mencoba untuk membedakan antara gambar asli dan gambar palsu.

Setelah mendefinisikan generator dan discriminator, kamu bisa melatih GAN dengan cara yang unik. GANs melibatkan pelatihan generator dan discriminator secara bersamaan, di mana generator mencoba membuat gambar palsu yang semakin baik untuk menipu discriminator, dan discriminator mencoba menjadi semakin baik dalam membedakan antara gambar asli dan palsu. GANs memiliki banyak potensi untuk berbagai aplikasi, termasuk pembuatan gambar dan video baru, peningkatan resolusi gambar, dan lainnya.

3.7 Transformers

3.7.1 Konsep Transformers

Transformers mengambil pendekatan yang berbeda dalam memodelkan urutan data. Differensiasi utamanya adalah penggunaan mekanisme yang disebut attention sebagai pengganti konvolusi atau pengulangan.

Mekanisme Attention
Mekanisme attention pada dasarnya mencoba mengukur bagaimana setiap item dalam suatu urutan berhubungan dengan item lain. Dalam konteks kalimat, ini berarti bahwa Transformer dapat mempelajari hubungan antar kata dalam kalimat tanpa peduli dengan jarak antara kata-kata tersebut. Ini berbeda dengan RNN dan LSTM yang harus memproses kalimat dari satu ujung ke ujung lainnya.

Model Transformer
Model Transformer terdiri dari dua bagian utama, yaitu enkoder dan dekoder.

1. Enkoder: Enkoder bertugas menerima input dan mengubahnya menjadi suatu representasi yang dapat dipahami oleh dekoder. Enkoder pada Transformer terdiri dari serangkaian lapisan identik. Setiap lapisan memiliki dua sub-lapisan, yaitu self-attention dan feed forward neural network. Dalam setiap sub-lapisan, terdapat juga residual connection dan layer normalization.
2. Dekoder: Dekoder menerima output dari enkoder dan menghasilkan output akhir. Dekoder pada Transformer memiliki struktur yang mirip dengan enkoder, tetapi ada tambahan sub-lapisan yang melakukan attention terhadap output dari enkoder.

3.7.2 Penerapan Transformers: Code Snippet dan Studi Kasus

Dalam Python, kita bisa memanfaatkan library HuggingFace's Transformers yang telah memaketkan model Transformer dan menawarkan interface yang mudah digunakan.

Berikut adalah contoh sederhana bagaimana menggunakan model Transformer untuk task text classification dengan dataset dummy:

```
!pip install transformers
!pip install torch

from transformers import BertTokenizer, BertForSequenceClassification
import torch

# Inisialisasi tokenizer dan model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model =
BertForSequenceClassification.from_pretrained('bert-base-uncased')

# Contoh kalimat
sentences = ['This is a positive sentence.', 'This is a negative
sentence.']

# Tokenize kalimat
inputs = tokenizer(sentences, padding=True, truncation=True,
return_tensors='pt')

# Forward pass melalui model
outputs = model(**inputs)

# Hasil logits
logits = outputs.logits

# Softmax untuk mendapatkan probabilitas
probs = torch.nn.functional.softmax(logits, dim=-1)
print(probs)
```

Output:

```
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
tensor([[0.5054, 0.4946],
        [0.5035, 0.4965]], grad_fn=<SoftmaxBackward0>)
```

Tips dan Trik:

1. Banyaknya Data: Semakin banyak data yang kamu miliki, semakin baik model transformer akan bekerja. Model-model ini sangat kompleks dan memiliki banyak parameter, sehingga mereka memerlukan banyak data untuk belajar.
2. Pemilihan Model: Ada banyak variasi model transformer yang telah dilatih sebelumnya seperti BERT, GPT-2, RoBERTa, dan T5. Setiap model memiliki kelebihan dan kekurangan tersendiri, jadi pilihlah model yang paling cocok untuk tugas spesifik kamu.

3. Penggunaan Library: Library seperti Hugging Face sangat berguna untuk bekerja dengan model transformer. Mereka menyediakan banyak fungsi bantuan dan model yang telah dilatih sebelumnya yang dapat memudahkan pekerjaanmu.
4. Fine-Tuning: Ketika menggunakan model transformer yang telah dilatih sebelumnya, sangat penting untuk melakukan fine-tuning model pada tugas spesifik kamu. Ini berarti melatih ulang beberapa lapisan akhir model pada data kamu, yang dapat meningkatkan kinerja model.
5. Perhatikan Dimensi Input: Model transformer memiliki batasan maksimum pada panjang input. Untuk BERT, batasannya adalah 512 token. Jadi pastikan untuk memotong atau memadatkan teks kamu agar sesuai.

Bab 4. Deep Learning Tools dan Libraries

4.1 Tensorflow

TensorFlow adalah library open-source untuk pembelajaran mesin dan deep learning yang dikembangkan oleh Google Brain Team. TensorFlow mendukung berbagai jenis algoritma dan metode pembelajaran mesin, tetapi paling dikenal untuk kemampuannya dalam memfasilitasi dan mempercepat pengembangan dan penerapan model deep learning.

Pengenalan ke TensorFlow
TensorFlow memanfaatkan grafik komputasi dimana node dalam grafik mewakili operasi matematika, sementara edge mewakili tensor (array multidimensi) yang ditransfer antara node. Konsep ini memungkinkan TensorFlow untuk melakukan komputasi paralel yang efisien dan mendistribusikan proses di banyak GPU atau bahkan mesin.

TensorFlow 2.0 dan Eager Execution
Dengan peluncuran TensorFlow 2.0, Google memperkenalkan konsep Eager Execution. Ini berarti bahwa operasi TensorFlow dijalankan segera saat mereka didefinisikan, bukan membangun grafik untuk dijalankan nanti. Ini membuat TensorFlow lebih intuitif dan lebih mudah digunakan, terutama bagi mereka yang baru memulai dengan library.

4.1.1 Menggunakan TensorFlow untuk Deep Learning

TensorFlow menyediakan berbagai API tingkat tinggi (seperti Keras) dan tingkat rendah yang memungkinkan kamu untuk membangun dan melatih model deep learning.

Contoh sederhana penggunaan TensorFlow untuk membangun model neural network sederhana untuk klasifikasi gambar pada dataset Fashion MNIST adalah sebagai berikut:

```
# Import TensorFlow
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

# Load the Fashion-MNIST dataset
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

# Preprocess the data
train_images = train_images / 255.0
test_images = test_images / 255.0
```

```
# Build the model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

# Compile the model
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=5)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images,  test_labels,
verbose=2)

print('\nTest accuracy:', test_acc)
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 2s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 1s 0us/step
Epoch 1/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.4973 - accuracy: 0.8260
Epoch 2/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3724 - accuracy: 0.8658
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3342 - accuracy: 0.8785
Epoch 4/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.3098 - accuracy: 0.8856
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2924 - accuracy: 0.8921
313/313 - 1s - loss: 0.3459 - accuracy: 0.8775 - 659ms/epoch - 2ms/step

Test accuracy: 0.8774999976158142
```

Dalam contoh ini, kita pertama-tama memuat dataset Fashion MNIST menggunakan fungsi bawaan TensorFlow. Kemudian, kita memproses data dengan membagi setiap piksel gambar dengan 255.0, sehingga nilainya menjadi antara 0 dan 1.

Kemudian kita membangun model neural network dengan dua layer Dense. Layer Flatten mengubah format gambar dari array dua dimensi (28 x 28 piksel) menjadi array satu dimensi (28 * 28 = 784 piksel).

Pada saat melakukan kompilasi model, kita menentukan optimizer (adam), fungsi loss (SparseCategoricalCrossentropy), dan metrik (accuracy) yang akan digunakan.

Akhirnya, kita melatih model dengan memanggil fungsi fit pada objek model dan memberikan data pelatihan dan label pelatihan, serta jumlah epoch (iterasi melalui seluruh dataset). Setelah melatih model, kita mengevaluasi akurasi model pada data uji.

Tips dan Trik:

1. Menggunakan GPU: TensorFlow dirancang untuk dapat dijalankan secara efisien di CPU dan GPU. Untuk mengaktifkan eksekusi GPU, pastikan kamu memiliki CUDA dan cuDNN yang diperlukan jika kamu menggunakan NVIDIA GPU. TensorFlow secara otomatis menggunakan GPU jika tersedia.
2. Memilih tingkat API yang tepat: TensorFlow menyediakan berbagai tingkat API, dari tingkat tinggi seperti Keras, hingga API tingkat rendah yang memberikan kontrol lebih besar. Biasanya, lebih mudah dan lebih cepat untuk mulai dengan API tingkat tinggi, dan kemudian pindah ke API tingkat rendah jika kamu membutuhkan kontrol lebih besar.

3. Penggunaan tf.data untuk memuat data: Modul tf.data di TensorFlow menyediakan kelas dan fungsi yang memudahkan muatan dan pra-pemrosesan data. Itu juga memungkinkan kamu untuk membuat pipeline data yang efisien dan mudah digunakan.
4. Pemantauan dengan TensorBoard: TensorBoard adalah alat visualisasi yang disertakan dengan TensorFlow yang memungkinkan kamu untuk memantau proses pelatihan model dalam waktu nyata.
5. Simpan dan muat model dengan tf.saved_model atau tf.keras.models.save_model: TensorFlow menyediakan fungsi untuk menyimpan dan memuat model, yang memungkinkan kamu untuk melanjutkan pelatihan dari mana kamu berhenti, atau menggunakan model yang telah dilatih sebelumnya.

Dalam konteks TensorFlow dan deep learning, berikut adalah beberapa parameter dan metode penting:

tf.keras.Sequential

tf.keras.Sequential adalah class untuk membuat model neural network secara sequential di TensorFlow. Dalam contoh kode yang disediakan sebelumnya, kita menggunakan ini untuk membuat model kita.

Parameter penting yang dapat diterima oleh tf.keras.Sequential adalah list layer yang ingin kita tambahkan ke model kita. Setiap elemen dalam list harus menjadi objek layer yang dapat diinstansiasi.

tf.keras.layers.Dense

tf.keras.layers.Dense adalah layer paling standar dan sering digunakan dalam neural network. Layer ini terhubung secara penuh dengan layer sebelumnya, yang berarti setiap neuron di layer ini terhubung ke semua neuron di layer sebelumnya.

Parameter penting dalam tf.keras.layers.Dense:
units: jumlah neuron di layer ini.
activation: fungsi aktivasi yang digunakan. Ini bisa berupa string yang mewakili fungsi aktivasi (mis. 'relu', 'sigmoid', 'softmax') atau bisa juga objek fungsi.
model.compile

Fungsi compile digunakan untuk mengonfigurasi proses pembelajaran model. Ada tiga parameter penting:
optimizer: ini menentukan algoritma optimasi yang akan digunakan. Bisa berupa string (mis. 'adam', 'sgd') atau instance dari class tf.keras.optimizers.
loss: ini menentukan fungsi kerugian yang digunakan. Bisa berupa string (mis. 'mean_squared_error', 'categorical_crossentropy') atau bisa juga objek fungsi.
metrics: ini menentukan metrik yang digunakan untuk mengukur kualitas model selama pelatihan dan pengujian. Biasanya diisi dengan 'accuracy'.

model.fit
model.fit adalah fungsi yang digunakan untuk melatih model selama jumlah epoch yang ditentukan.

Parameter penting:
x: data input.
y: data target.
epochs: jumlah epoch, yaitu berapa kali model melalui seluruh dataset.
batch_size: jumlah sampel per pembaruan gradien. Jika tidak ditentukan, batch_size akan default ke 32.

model.evaluate
model.evaluate adalah fungsi yang digunakan untuk mengevaluasi model.

Parameter penting:
x: data input.
y: data target.
batch_size: jumlah sampel per pembaruan gradien. Jika tidak ditentukan, batch_size akan default ke 32.
Selain metode-metode di atas, TensorFlow juga menyediakan berbagai fungsi dan operasi lainnya yang membantu dalam proses pembelajaran mesin dan deep learning, termasuk operasi matematika dasar, operasi tensor, dan sebagainya. Dianjurkan untuk selalu merujuk ke dokumentasi resmi TensorFlow untuk penjelasan lengkap dan rinci tentang fungsi dan metode yang disediakan oleh library.

4.2 Keras

Keras adalah library machine learning berbasis Python yang bersifat open-source. Fokus utamanya adalah memudahkan dalam pembuatan dan pengujian model deep learning. Keras berfungsi sebagai interface untuk library TensorFlow, yang berarti Keras memanfaatkan kekuatan TensorFlow dan memberikan kemudahan dalam membuat dan menguji model neural network.

Berikut ini adalah panduan sederhana tentang bagaimana menggunakan Keras untuk deep learning.

4.2.1 Membuat Model Neural Network dengan Keras

Untuk membuat model neural network dengan Keras, kamu perlu melakukan beberapa langkah:

Import Keras

```
from keras.models import Sequential
from keras.layers import Dense
```

Definisikan Model

Kamu perlu mendefinisikan model kamu. Dalam hal ini, kami akan menggunakan model Sequential, yang memungkinkan kamu untuk mendefinisikan model layer by layer.

```
model = Sequential()
```

Tambahkan Layer

Sekarang, kamu perlu menambahkan layer ke dalam model. Misalnya, jika ingin menambahkan dense layer dengan 32 unit dan fungsi aktivasi ReLU, kamu bisa melakukannya seperti ini:

```
model.add(Dense(32, activation='relu'))
```

Definisikan Fungsi Loss dan Optimizer

Selanjutnya, kamu perlu menentukan fungsi loss dan optimizer untuk model kamu. Kita akan menggunakan categorical_crossentropy sebagai fungsi loss dan adam sebagai optimizer.

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Latih Model

Terakhir, kamu bisa melatih model dengan menggunakan fungsi fit. Kamu perlu memberikan data pelatihan dan label, serta menentukan jumlah epoch.

```
model.fit(x_train, y_train, epochs=10)
```

Dengan cara ini, kamu dapat membuat model neural network sederhana dengan Keras.

4.2.2 Parameter dan Metode dalam Keras

Berikut adalah beberapa parameter dan metode yang sering digunakan dalam Keras:

1. **model.compile:** Digunakan untuk mengkonfigurasi model untuk pelatihan. Parameter yang penting adalah loss (fungsi kerugian yang digunakan), optimizer (algoritma optimasi yang digunakan), dan metrics (metrik evaluasi model).
2. **model.fit:** Digunakan untuk melatih model selama jumlah epoch yang ditentukan. Parameter yang penting adalah x (data input), y (data target), epochs (jumlah epoch), dan batch_size (jumlah sampel per pembaruan gradien).
3. **model.evaluate:** Digunakan untuk mengevaluasi kinerja model. Parameter yang penting adalah x (data input), y (data target), dan batch_size (jumlah sampel per pembaruan gradien).
4. **model.predict:** Digunakan untuk membuat prediksi menggunakan model. Parameter yang penting adalah x (data input) dan batch_size (jumlah sampel per pembaruan gradien).
5. **Dense:** Dense adalah layer standar yang bekerja dengan baik pada berbagai jenis masalah. Parameter yang penting adalah units (jumlah unit output), activation (fungsi aktivasi yang digunakan), dan input_dim (dimensi input, yang hanya perlu ditentukan untuk layer pertama).
6. **Conv2D:** Conv2D adalah layer konvolusi yang digunakan untuk masalah yang melibatkan data berdimensi tinggi, seperti gambar. Parameter yang penting adalah filters (jumlah filter output), kernel_size (ukuran kernel konvolusi), dan activation (fungsi aktivasi yang digunakan).
7. **MaxPooling2D:** MaxPooling2D adalah layer pooling yang digunakan untuk mengurangi dimensi spasial dari data. Parameter yang penting adalah pool_size (ukuran pooling).

4.2.3 Tips dan Trik dalam Menggunakan Keras

Berikut adalah beberapa tips dan trik yang dapat membantu kamu menggunakan Keras lebih efisien:

1. **Gunakan Model Callback:** Keras menyediakan beberapa callback yang bisa kamu gunakan untuk mendapatkan visibilitas lebih baik tentang apa yang terjadi di dalam model selama pelatihan.
2. **Eksperimen dengan Berbagai Jenis Layer:** Keras memiliki berbagai jenis layer yang bisa kamu gunakan, seperti convolutional layers, pooling layers, dropout layers, dan sebagainya. Jangan takut untuk bereksperimen dengan berbagai jenis layer untuk melihat apa yang bekerja terbaik untuk masalah kamu.

3. **Pilih Fungsi Aktivasi yang Tepat:** Fungsi aktivasi yang kamu pilih untuk layer kamu dapat mempengaruhi hasil yang kamu dapatkan. Biasanya, fungsi aktivasi ReLU bekerja dengan baik pada kebanyakan masalah, tetapi kamu mungkin ingin mencoba yang lain seperti sigmoid atau tanh untuk masalah tertentu.
4. **Gunakan Early Stopping:** Early stopping adalah teknik yang menghentikan pelatihan jika model tidak memperbaiki performanya lagi. Ini bisa membantu kamu menghindari overfitting dan menghemat waktu.
5. **Tune Hyperparameters:** Kamu mungkin perlu melakukan tuning hyperparameter, seperti learning rate, jumlah unit di setiap layer, dan sebagainya, untuk mendapatkan hasil terbaik.

Keras adalah alat yang sangat kuat untuk deep learning, dan dengan menggunakan tips dan trik ini, kamu dapat memanfaatkannya secara maksimal. Jangan lupa untuk selalu merujuk ke dokumentasi resmi Keras untuk informasi lebih lanjut dan lebih detail tentang fungsi dan metode yang disediakan oleh library.

Berikut adalah contoh penggunaan Keras untuk membuat dan melatih model neural network sederhana untuk mengklasifikasikan digit tulisan tangan dari dataset MNIST.

```
# Import required libraries
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess data
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# Define model
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
```

```
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

# Compile model
model.compile(loss='categorical_crossentropy', optimizer=RMSprop(),
metrics=['accuracy'])

# Train model
model.fit(x_train, y_train, batch_size=128, epochs=10, verbose=1,
validation_data=(x_test, y_test))

# Evaluate model
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Output:

```
Epoch 1/10
469/469 [==============================] - 3s 5ms/step - loss: 0.2453 - accuracy: 0.9244 - val_loss: 0.1018 - val_accuracy: 0.9670
Epoch 2/10
469/469 [==============================] - 2s 5ms/step - loss: 0.1030 - accuracy: 0.9681 - val_loss: 0.0872 - val_accuracy: 0.9740
Epoch 3/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0770 - accuracy: 0.9767 - val_loss: 0.0736 - val_accuracy: 0.9792
Epoch 4/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0597 - accuracy: 0.9819 - val_loss: 0.0756 - val_accuracy: 0.9799
Epoch 5/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0490 - accuracy: 0.9850 - val_loss: 0.0770 - val_accuracy: 0.9792
Epoch 6/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0428 - accuracy: 0.9873 - val_loss: 0.0876 - val_accuracy: 0.9777
Epoch 7/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0384 - accuracy: 0.9886 - val_loss: 0.0809 - val_accuracy: 0.9817
Epoch 8/10
469/469 [==============================] - 3s 5ms/step - loss: 0.0346 - accuracy: 0.9896 - val_loss: 0.0878 - val_accuracy: 0.9824
Epoch 9/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0306 - accuracy: 0.9908 - val_loss: 0.0876 - val_accuracy: 0.9812
Epoch 10/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0286 - accuracy: 0.9918 - val_loss: 0.0796 - val_accuracy: 0.9856
Test loss: 0.07962159067392349
Test accuracy: 0.9855999946594238
```

Load MNIST dataset: Ini memuat dataset MNIST yang berisi gambar digit tulisan tangan.

Preprocess data: Data diproses sebelum dilatih. Gambar direpresentasikan sebagai array dua dimensi. Mereka diubah menjadi array satu dimensi dan dinormalisasi sehingga nilai pikselnya berada dalam rentang 0-1.

Convert class vectors to binary class matrices: Label diproses menjadi format yang dapat digunakan oleh model. Ini disebut one-hot encoding.

Define model: Model neural network dibuat menggunakan model Sequential. Model ini memiliki tiga layer Dense (alias fully connected), di mana dua layer pertama menggunakan fungsi aktivasi ReLU dan layer terakhir menggunakan softmax.

Compile model: Model dikompilasi dengan fungsi loss (categorical_crossentropy), optimizer (RMSprop), dan metrik (accuracy) yang ditentukan.

Train model: Model dilatih menggunakan data dan label latihan, dengan ukuran batch 128 dan 10 epoch. Data dan label tes digunakan sebagai data validasi.

Evaluate model: Model dievaluasi menggunakan data dan label tes, dan loss serta akurasi dicetak.

4.3 PyTorch

PyTorch adalah library open source yang sangat populer untuk machine learning dan khususnya deep learning. PyTorch menyediakan dua fitur utama:

Tensor komputasi, seperti array NumPy, tetapi dapat berjalan di GPU
Fungsionalitas deep learning otomatis dan berbeda untuk desain dan pelatihan jaringan saraf.

Berikut beberapa hal penting untuk dipahami saat menggunakan PyTorch:

1. **Tensors:** Ini adalah struktur data dasar di PyTorch dan mirip dengan array NumPy. Mereka dapat berjalan pada GPU, yang memungkinkan percepatan komputasi yang signifikan.
2. **Autograd:** PyTorch menyediakan paket bernama autograd untuk otomatisasi perhitungan gradient. Ini sangat berguna saat melatih model neural network.
3. **nn Module:** PyTorch menyediakan modul nn yang berisi kelas dan fungsi untuk membangun jaringan saraf. Ini memungkinkan kamu untuk mendefinisikan layer dan model dengan cara yang jelas dan intuitif.
4. **Optimizers:** Modul torch.optim menyediakan algoritma optimasi yang umum digunakan dalam pelatihan jaringan saraf, seperti SGD, RMSProp, dan Adam.

Berikut adalah contoh skrip menggunakan PyTorch untuk melatih model jaringan saraf sederhana pada dataset Iris:

```
import torch
from torch import nn, optim
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

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

```
# Preprocessing
scaler = StandardScaler()
X = scaler.fit_transform(X)

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

# Convert to PyTorch tensors
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.LongTensor(y_train)
y_test = torch.LongTensor(y_test)

# Define the network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(4, 100)
        self.fc2 = nn.Linear(100, 50)
        self.fc3 = nn.Linear(50, 3)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Instantiate the network, loss function and optimizer
model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    optimizer.zero_grad()
    out = model(X_train)
    loss = criterion(out, y_train)
    loss.backward()
    optimizer.step()

# Evaluation
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    outputs = model(X_test)
```

```
    _, predicted = torch.max(outputs.data, 1)
    total += y_test.size(0)
    correct += (predicted == y_test).sum().item()

print('Test Accuracy: {} %'.format(100 * correct / total))
```

Output:

```
Test Accuracy: 93.33333333333333 %
```

Pada skrip di atas, kita melakukan hal-hal berikut:

1. Mengimpor library yang diperlukan, termasuk PyTorch dan beberapa fungsi dari Scikit-Learn.
2. Memuat dan memproses dataset Iris. Dataset dibagi menjadi set latihan dan tes, dan fitur-fiturnya dinormalisasi.
3. Mendefinisikan model jaringan saraf menggunakan modul nn. Model ini memiliki dua layer tersembunyi dan output layer dengan tiga neuron (satu untuk setiap kelas dalam dataset Iris).
4. Mendefinisikan fungsi loss (CrossEntropyLoss) dan optimizer (SGD).
5. Melakukan loop pelatihan, di mana gradient dihitung dan bobot model diperbarui.
6. Mengevaluasi model pada set tes dan mencetak akurasi.

Tips dan trik:

1. Jika kamu memiliki GPU, pastikan untuk memanfaatkan kemampuan PyTorch untuk melakukan komputasi pada GPU. Ini bisa sangat mempercepat waktu pelatihan model.
2. Gunakan autograd untuk otomatisasi perhitungan gradient. Ini akan membuat kode kamu lebih rapi dan mudah untuk dit-debug.
3. Cobalah berbagai optimizer dan lihat bagaimana mereka mempengaruhi kinerja model. Kadang-kadang, perubahan kecil dalam optimizer atau learning rate bisa membuat perbedaan yang besar dalam kinerja model.
4. Jangan lupa untuk memanggil model.eval() sebelum mengevaluasi model. Ini akan menonaktifkan layer seperti dropout dan batch normalization yang seharusnya tidak aktif saat evaluasi.
5. Jika model kamu tidak belajar, cobalah cek apakah gradient mengalir dengan benar melalui jaringan. Kamu dapat melakukan ini dengan mencetak gradient untuk beberapa parameter setelah memanggil loss.backward().

Ketika menggunakan PyTorch untuk deep learning, ada banyak parameter dan metode yang digunakan. Berikut adalah beberapa yang paling penting:

Parameter dalam modul nn

1. **in_features dan out_features:** Parameter ini mendefinisikan jumlah input dan output unit dalam layer linear (nn.Linear).

2. **num_features:** Ini digunakan dalam layer seperti batch normalization (nn.BatchNorm2d) dan menentukan jumlah fitur dalam input.
3. **kernel_size, stride, dan padding:** Ini adalah parameter yang digunakan dalam konvolusi (nn.Conv2d) dan pooling layers (nn.MaxPool2d), dan menentukan ukuran dan langkah dari kernel, serta padding yang digunakan di sekitar input.

Metode dalam modul nn

1. **forward(x):** Ini adalah metode yang harus kamu definisikan saat membuat kelas jaringan sarafmu sendiri yang mewarisi dari nn.Module. Metode ini mendefinisikan bagaimana data bergerak maju melalui jaringan.
2. **zero_grad():** Metode ini mengatur semua gradient dalam model menjadi nol. Ini biasanya dipanggil di awal langkah pelatihan sebelum melakukan backpropagation.
3. **backward():** Metode ini melakukan backpropagation dan menghitung semua gradient. Biasanya dipanggil setelah menghitung loss.
4. **step():** Metode ini melakukan langkah pembaruan pada optimizer. Biasanya dipanggil setelah backpropagation.

Parameter dalam modul optim

1. **params:** Ini adalah parameter yang akan diperbarui oleh optimizer. Biasanya, ini adalah parameter model yang kamu dapatkan dengan memanggil model.parameters().
2. **lr:** Ini adalah learning rate, yang menentukan seberapa besar langkah yang diambil saat melakukan pembaruan parameter.
3. **momentum:** Parameter ini digunakan dalam optimizer seperti SGD untuk membantu menghindari jebakan lokal dan mempercepat pelatihan.

Metode dalam modul optim

1. **zero_grad():** Seperti dalam nn.Module, metode ini mengatur semua gradient menjadi nol.
2. **step():** Seperti dalam nn.Module, metode ini melakukan langkah pembaruan pada optimizer.

Harap diingat bahwa ini adalah parameter dan metode yang paling umum digunakan dalam konteks deep learning dengan PyTorch. Ada banyak lagi fitur yang disediakan oleh PyTorch yang dapat membantu kamu dalam berbagai kasus penggunaan lainnya. Selalu periksa dokumentasi PyTorch jika kamu tidak yakin tentang apa yang dilakukan oleh parameter atau metode tertentu.

4.4 Scikit-learn

Scikit-learn adalah library Python yang menyediakan berbagai alat untuk machine learning dan statistical modeling termasuk klasifikasi, regresi, clustering dan dimensionality

reduction. Meskipun tidak secara khusus dibuat untuk deep learning, Scikit-learn mendukung penerapan neural networks dengan Multi-layer Perceptron (MLP).

Berikut adalah cheatsheet untuk Scikit-learn dalam konteks deep learning:

Import Library

```
from sklearn.neural_network import MLPClassifier, MLPRegressor
```

Membuat Model

```
# Untuk kasus klasifikasi
clf = MLPClassifier(hidden_layer_sizes=(100,100,100), activation='relu',
solver='adam', max_iter=500)

# Untuk kasus regresi
reg = MLPRegressor(hidden_layer_sizes=(100,100,100), activation='relu',
solver='adam', max_iter=500)
```

Parameter penting dalam MLP Scikit-learn:

hidden_layer_sizes: Tuple, panjang n_layers - 2, default=(100,). Elemen i merepresentasikan jumlah neuron dalam hidden layer ke-i.

activation: {'identity', 'logistic', 'tanh', 'relu'}, default='relu'. Fungsi aktivasi untuk hidden layer.

solver: {'lbfgs', 'sgd', 'adam'}, default='adam'. Solver untuk optimasi bobot.

max_iter: int, default=200. Maksimum iterasi.

Training Model

```
# Untuk kasus klasifikasi
clf.fit(X_train, y_train)

# Untuk kasus regresi
reg.fit(X_train, y_train)
```

Prediksi

```
# Untuk kasus klasifikasi
predictions = clf.predict(X_test)

# Untuk kasus regresi
```

```
predictions = reg.predict(X_test)
```

Evaluasi Model

```
from sklearn.metrics import accuracy_score, mean_squared_error

# Klasifikasi
accuracy = accuracy_score(y_test, predictions)

# Regresi
mse = mean_squared_error(y_test, predictions)
```

Berikut ini adalah contoh script menggunakan Scikit-learn untuk membangun dan melatih model Multi-Layer Perceptron (MLP) untuk tugas klasifikasi.

```
# Import Libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

# Load iris dataset
iris = load_iris()

# Prepare data
X = iris.data
y = iris.target

# Split data into training and test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create an MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(100,), max_iter=300, activation
= 'relu',solver='adam',random_state=1)

# Train the model
clf.fit(X_train, y_train)

# Make predictions
```

```
y_pred = clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy}')
```

Output:

```
Accuracy: 1.0
```

Penjelasan:

Script di atas melibatkan beberapa tahapan umum dalam proses machine learning:

1. Mempersiapkan Data: Kami menggunakan dataset iris yang tersedia dalam Scikit-learn. Dataset ini kemudian dibagi menjadi training set dan test set.
2. Normalisasi Data: Kami menggunakan StandardScaler untuk menormalkan fitur, yang merupakan praktek umum dalam deep learning untuk membantu model belajar lebih efisien.
3. Membangun Model: Kami membuat MLPClassifier dengan satu hidden layer yang terdiri dari 100 neuron. Fungsi aktivasi yang digunakan adalah 'relu' dan solver untuk optimasi bobot adalah 'adam'. Maksimum iterasi ditetapkan pada 300.
4. Melatih Model: Menggunakan metode .fit() untuk melatih model dengan data training.
5. Membuat Prediksi: Setelah model dilatih, kita dapat menggunakan metode .predict() untuk membuat prediksi pada data test.
6. Evaluasi Model: Akhirnya, kami menggunakan fungsi accuracy_score() untuk menghitung akurasi model pada data test.

Tips dan Trik

1. Untuk data dengan fitur dalam skala yang berbeda, sangat penting untuk melakukan normalisasi atau standardisasi sebelum melatih model.
2. Meningkatkan max_iter dapat membantu model untuk konvergen, tetapi juga dapat meningkatkan waktu pelatihan.
3. Jika model overfitting, kamu bisa mencoba mengurangi jumlah hidden layers atau jumlah neuron di dalamnya.
4. Kamu dapat menggunakan cross-validation dan grid search untuk mencari kombinasi parameter terbaik.

Dengan memahami fungsi dasar dari Scikit-learn dan MLP, kamu sudah siap untuk menggunakan library ini untuk mengeksplorasi dunia deep learning lebih jauh. Meski tidak fleksibel dan kuat seperti Tensorflow atau PyTorch, Scikit-learn memberikan pendekatan

yang lebih mudah dan langsung untuk memahami konsep dasar neural networks dan deep learning.

Bab 5. Deep Learning dalam Pengolahan Gambar dan Vision

5.1 Object Detection

Object Detection adalah salah satu cabang dalam computer vision yang berfokus pada identifikasi dan lokasi objek spesifik dalam sebuah gambar atau video sequence. Dalam object detection, kami tidak hanya mengklasifikasikan objek (seperti dalam Image Classification), tetapi juga menentukan bounding box atau lokasi spesifik objek tersebut dalam gambar.

Ada berbagai metode dan algoritma yang digunakan dalam object detection, di antaranya:

1. R-CNN (Regions with CNN): Algoritma ini pertama-tama mencari region proposal atau wilayah-wilayah yang mungkin mengandung objek, kemudian menggunakan Convolutional Neural Networks (CNN) untuk mengklasifikasikan objek tersebut.
2. Fast R-CNN: Perkembangan dari R-CNN, Fast R-CNN menggunakan teknik yang disebut ROI Pooling untuk mengatasi masalah kecepatan dan efisiensi dalam R-CNN.
3. Faster R-CNN: Sebuah peningkatan lagi dari Fast R-CNN, Faster R-CNN menggantikan pencarian region proposal dengan network yang disebut Region Proposal Network (RPN), mempercepat proses lebih jauh.
4. YOLO (You Only Look Once): Berbeda dengan R-CNN dan variasinya, YOLO menggunakan pendekatan single-shot, artinya ia melakukan deteksi dan klasifikasi dalam satu kali "lihat" ke gambar, membuatnya sangat cepat.
5. SSD (Single Shot MultiBox Detector): Mirip dengan YOLO, SSD juga menggunakan pendekatan single-shot, tetapi dengan cara yang berbeda dalam hal prediksi bounding box.

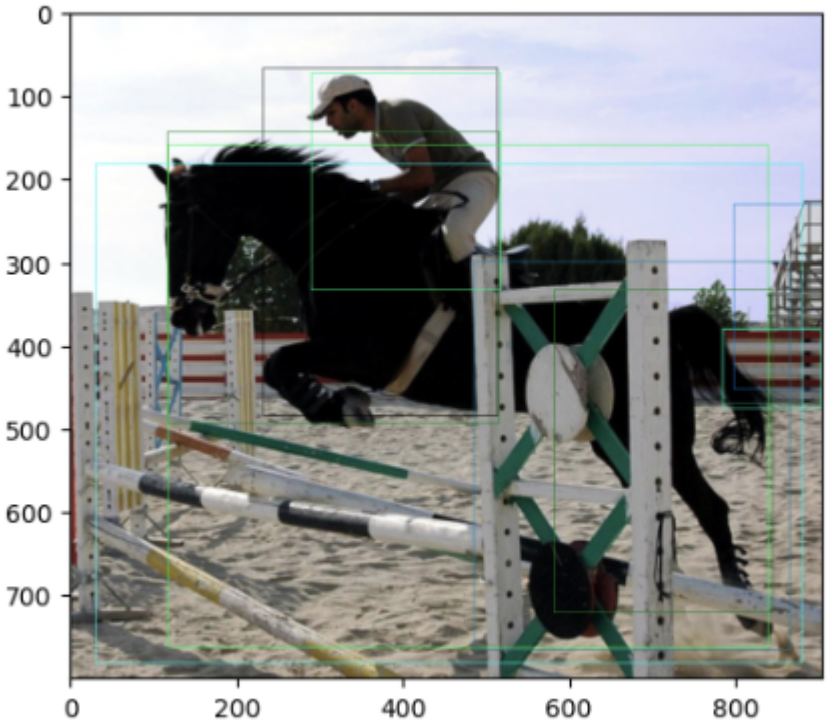
Contoh Script: Object Detection dengan YOLO
Berikut adalah contoh script yang menggunakan YOLO untuk object detection pada sample dataset. Dalam contoh ini, kami menggunakan library PyTorch dan torchvision untuk memuat model dan dataset.

```
import torch
import torchvision.transforms as T
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.datasets import VOCDetection
from torchvision.utils import draw_bounding_boxes
import matplotlib.pyplot as plt

# Load the pre-trained model
model = fasterrcnn_resnet50_fpn(pretrained=True)
model.eval()
```

```
# Load the dataset
dataset = VOCDetection(root='.', year='2012', image_set='train',
download=True)
transform = T.Compose([T.Resize(800), T.ToTensor()])

def detect_objects(image):
    # Transform the image and add a batch dimension
    image_t = transform(image).unsqueeze(0)

    # Forward pass
    with torch.no_grad():
        predictions = model(image_t)

    # Get the bounding boxes and labels
    boxes = predictions[0]['boxes']
    labels = predictions[0]['labels']

    return boxes, labels, image_t

# Select an image from the dataset
image, _ = dataset[0]

# Detect objects in the image
boxes, labels, image_t = detect_objects(image)

# Convert the image tensor to uint8
image_t = (image_t * 255).byte()

# Draw the bounding boxes
image_with_boxes = draw_bounding_boxes(image_t.squeeze(0), boxes)

# Display the image
plt.imshow(image_with_boxes.permute(1, 2, 0))
plt.show()
```

Output:

```
Using downloaded and verified file: ./VOCtrainval_11-May-2012.tar
Extracting ./VOCtrainval_11-May-2012.tar to .
```



Berikut ini adalah penjelasan detil dari script Python di atas:

Pertama, script ini mengimpor beberapa modul yang diperlukan:

1. torch: ini adalah library PyTorch, yang digunakan untuk komputasi tensor dan deep learning.
2. torchvision.transforms: ini adalah modul yang menyediakan beberapa transformasi umum yang bisa kita gunakan untuk preprocessing gambar.
3. fasterrcnn_resnet50_fpn dari torchvision.models.detection: ini adalah model Faster R-CNN dengan backbone ResNet-50 dan Feature Pyramid Network (FPN) yang telah dilatih sebelumnya.
4. VOCDetection dari torchvision.datasets: ini adalah dataset Pascal VOC, yang digunakan untuk deteksi objek.
5. draw_bounding_boxes dari torchvision.utils: ini adalah fungsi yang digunakan untuk menggambar bounding box pada gambar.
6. matplotlib.pyplot: ini adalah library untuk membuat plot dan visualisasi.
7. Kemudian, script ini memuat model Faster R-CNN yang telah dilatih sebelumnya dan mengubahnya ke mode evaluasi dengan model.eval().

Selanjutnya, script ini memuat dataset Pascal VOC tahun 2012 untuk set pelatihan dan mendownloadnya jika belum ada.

Script ini mendefinisikan fungsi detect_objects yang melakukan beberapa hal:

1. Mengubah ukuran gambar dan mengubahnya menjadi tensor menggunakan transformasi yang telah didefinisikan sebelumnya.
2. Melakukan forward pass melalui model untuk mendapatkan prediksi.
3. Mengambil bounding box dan label dari prediksi.
4. Script ini memilih gambar pertama dari dataset dan mendeteksi objek di dalamnya menggunakan fungsi detect_objects.
5. Setelah itu, script ini mengubah tensor gambar menjadi format uint8 dan menggambar bounding box pada gambar.
6. Terakhir, menampilkan gambar dengan bounding box menggunakan matplotlib.

Tips dan Trik dalam Object Detection

1. Pilih Model yang Tepat: Pilihan model akan sangat dipengaruhi oleh aplikasi yang kamu buat. Misalnya, jika kamu membutuhkan deteksi objek real-time, maka kamu mungkin ingin menggunakan model seperti SSD atau YOLO yang lebih cepat. Namun, jika keakuratan adalah yang paling penting, model seperti Faster R-CNN mungkin lebih sesuai.
2. Data Training yang Variatif: Pastikan bahwa data training kamu mencakup semua variasi objek yang ingin kamu deteksi. Ini termasuk variasi dalam pencahayaan, orientasi, ukuran, dan pose.
3. Data Augmentation: Teknik augmentasi data, seperti flipping, rotation, zooming, dll., dapat digunakan untuk meningkatkan jumlah dan variasi data training, yang akan meningkatkan performa model.
4. Fine-tuning: Jika dataset kamu tidak cukup besar, menggunakan model pre-trained dan melakukan fine-tuning dapat memberikan hasil yang baik.

5.2 Image Classification

Image classification adalah proses komputer yang mengklasifikasikan gambar ke dalam salah satu kategori atau kelas tertentu. Teknik ini termasuk dalam bidang computer vision dan telah menjadi semakin populer dengan perkembangan deep learning.

Deep Learning untuk Image Classification

Deep learning memanfaatkan neural networks dengan banyak layer ("deep" merujuk ke kedalaman jaringan), yang mampu belajar representasi gambar yang kompleks dan abstrak. Convolutional Neural Networks (CNNs) adalah jenis deep learning yang paling sering digunakan untuk image classification.

Convolutional Neural Networks (CNNs)

CNNs adalah jenis neural network yang dirancang khusus untuk mengolah data berupa gambar. CNNs terdiri dari beberapa layer berbeda, termasuk:

1. **Convolutional layers:** Layer ini mengaplikasikan filter kecil ke bagian dari gambar, biasanya berukuran 3x3 atau 5x5 piksel. Filter ini belajar mengenali fitur-fitur lokal dalam gambar, seperti garis atau tepian.
2. **Pooling layers:** Layer ini mengurangi resolusi gambar dengan mengambil maksimum atau rata-rata dari setiap bagian gambar.
3. **Fully connected layers:** Setelah beberapa layer convolutional dan pooling, gambar telah diubah menjadi representasi abstrak yang kemudian diklasifikasikan oleh satu atau lebih layer fully connected.

Contoh Script

Berikut ini adalah contoh script untuk image classification menggunakan CNNs pada dataset CIFAR-10, sebuah dataset populer yang terdiri dari 60,000 gambar berwarna 32x32 dalam 10 kelas (setiap kelas memiliki 6000 gambar). Dataset ini telah dibagi menjadi 50,000 gambar untuk training dan 10,000 gambar untuk testing.

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

# Load CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0

# Define the model architecture
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32,
32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Add Dense layers on top
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))

# Compile and train the model
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))

history.history
```

Output:

```
Epoch 10/10
1563/1563 [==============================] - 9s 6ms/step - loss: 0.5857 - accuracy: 0.7936 - val_loss: 0.8980 - val_accuracy: 0.7119
{'loss': [1.511352300643921,
  1.1465224027633667,
  0.992251455783844,
  0.8872874975204468,
  0.8135891556739807,
  0.7529917359352112,
  0.7089433073997498,
  0.6609216928482056,
  0.619335412979126,
  0.5856723785400391],
 'accuracy': [0.452239990234375,
  0.5940799713134766,
  0.6513400077819824,
  0.6884199976921082,
  0.7124999761581421,
  0.7351599931716919,
  0.7491000294685364,
  0.7679200172424316,
  0.7795400023460388,
  0.7936000227928162],
 'val_loss': [1.2815639972686768,
  1.0704048871994019,
  0.9839610457420349,
  0.9136618971824646,
  0.8776664137840271,
  0.8997917175292969,
  0.909877359867096,
  0.8733499050140381,
  0.8612900376319885,
  0.8979704976081848],
 'val_accuracy': [0.5410000085830688,
  0.6190000176429749,
  0.6549000144004822,
  0.6822999715805054,
  0.6992999911308289,
  0.6872000098228455,
  0.6963000297546387,
  0.7005000114440918,
  0.7095000147819519,
  0.711899995803833]}
```

Penjelasan

1. Script ini dimulai dengan import library yang diperlukan.
2. Kemudian, dataset CIFAR-10 di-load dan dibagi menjadi data train dan data test.
3. Nilai pixel gambar dinormalisasi untuk berada di antara 0 dan 1.
4. CNN dibuat dengan tiga layer convolutional, diikuti oleh dua layer fully connected.
5. Model kemudian dikompilasi dengan optimizer 'adam' dan loss function 'SparseCategoricalCrossentropy'.
6. Akhirnya, model di-fit (dilatih) menggunakan data train dan data test sebagai data validasi.
7. Hasil dari proses ini adalah model CNN yang dapat mengklasifikasikan gambar ke dalam 10 kelas yang ada di dataset CIFAR-10.

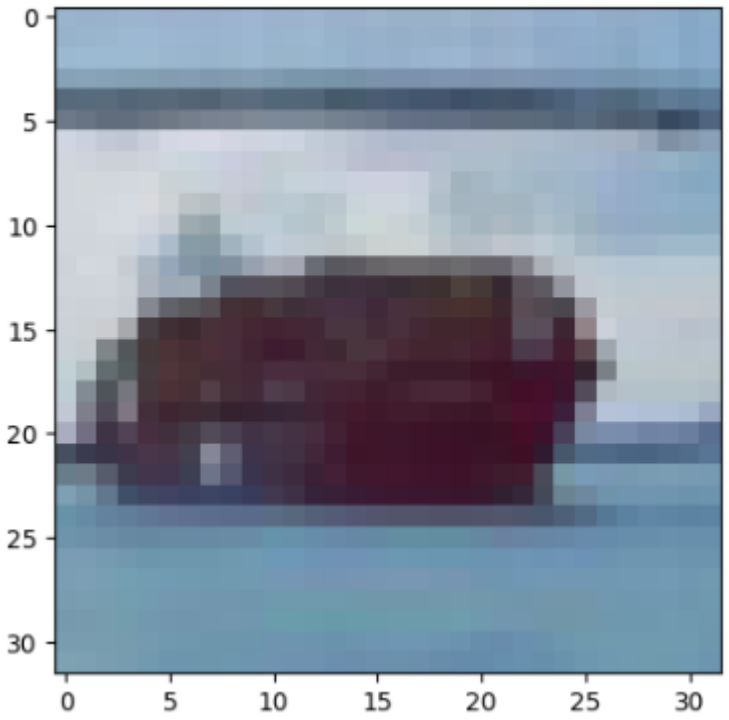
Untuk mendemonstrasikan klasifikasi gambar, kita bisa menggunakan gambar dari dataset CIFAR-10 itu sendiri. Berikut ini adalah contoh cara memprediksi label dari gambar dalam dataset tersebut menggunakan model yang telah dilatih.

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

```
# Choose a random image from the test dataset
i = np.random.randint(0, len(test_images))
img = test_images[i]

# Prepare the image to be used with the model
img_array = np.expand_dims(img, axis=0)

# Use the model to predict the image's class
pred = model.predict(img_array)
class_idx = np.argmax(pred[0])

# Define the CIFAR-10 classes
cifar10_classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog', 'frog', 'horse', 'ship', 'truck']

# Print the predicted and actual class of the image
print("Predicted class:", cifar10_classes[class_idx])
print("Actual class:", cifar10_classes[test_labels[i][0]])

# Display the image
plt.imshow(img)
plt.show()
```

Output:

```
1/1 [==============================] - 0s 409ms/step
Predicted class: ship
Actual class: ship
```



Tips dan Trik dalam Image Classification
Data Augmentation: Augmentasi data bisa sangat membantu untuk meningkatkan performa model dengan menciptakan variasi baru dari gambar yang ada. Ini bisa berupa rotasi, pergeseran, pembalikan, dan perubahan skala pada gambar.

1. Transfer Learning: Jika kamu memiliki dataset yang relatif kecil, metode transfer learning bisa sangat membantu. Ide dasarnya adalah menggunakan model yang telah dilatih pada dataset besar, dan kemudian menyesuaikannya pada dataset kamu. Ini biasanya dapat menghasilkan model yang lebih baik dibandingkan melatih model dari awal.
2. Hyperparameter Tuning: Ada banyak hyperparameter yang bisa diubah dalam CNN, seperti jumlah dan ukuran layer konvolusi, jenis fungsi aktivasi, dan jenis optimizer. Tuning hyperparameter bisa membantu kamu mendapatkan model terbaik.
3. Ensemble Models: Terkadang, menggunakan beberapa model dan menggabungkan prediksi mereka (dikenal sebagai 'ensembling') dapat memberikan hasil yang lebih baik daripada menggunakan satu model.

5.3. Image Segmentation

Image segmentation adalah proses membagi gambar digital menjadi beberapa segmen (set of pixels, juga dikenal sebagai image objects) untuk memudahkan analisis. Dalam konteks

deep learning dan computer vision, image segmentation digunakan untuk melabeli setiap piksel gambar dengan kelas label tertentu.

5.3.1. Apa Itu Image Segmentation?

Image segmentation adalah tugas membagi gambar menjadi beberapa segmen semantik; setiap segmen semantik ini harus memiliki interpretasi yang sama. Tujuan dari image segmentation adalah untuk memahami gambar dengan tingkat piksel, sehingga kita dapat lebih memahami objek yang ada di dalamnya dan hubungan antara mereka. Tugas ini digunakan dalam berbagai aplikasi, seperti pengenalan objek, pelacakan objek, dan bahkan dalam bidang medis untuk menentukan area patologis dalam scan medis.

Ada dua tipe utama dari image segmentation, yaitu semantic segmentation dan instance segmentation. Dalam semantic segmentation, setiap piksel gambar diberi label kelas tertentu (misalnya, "kucing", "anjing", "meja", dll.), dan semua objek dari kelas yang sama diberi label yang sama. Sementara itu, dalam instance segmentation, setiap instance objek diberi label unik; ini berarti jika ada dua objek dari kelas yang sama, mereka akan diberi label yang berbeda.

5.3.2. Image Segmentation Menggunakan Deep Learning

Dalam beberapa tahun terakhir, metode deep learning telah menjadi metode yang dominan untuk image segmentation. Salah satu model yang paling sering digunakan adalah U-Net, yang dirancang khusus untuk segmentasi gambar medis, tetapi dapat digunakan untuk berbagai jenis tugas segmentasi.

Berikut adalah contoh kode yang menunjukkan bagaimana melatih model U-Net untuk tugas segmentasi dengan menggunakan TensorFlow dan Keras:

```
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow_examples.models.pix2pix import pix2pix
import matplotlib.pyplot as plt

dataset, info = tfds.load('oxford_iiit_pet:3.*.*', with_info=True)

def normalize(input_image, input_mask):
  input_image = tf.cast(input_image, tf.float32) / 255.0
  return input_image, input_mask

@tf.function
def load_image_train(datapoint):
  input_image = tf.image.resize(datapoint['image'], (128, 128))
  input_mask = tf.image.resize(datapoint['segmentation_mask'], (128,
```

```
128))
  input_image, input_mask = normalize(input_image, input_mask)
  return input_image, input_mask

train = dataset['train'].map(load_image_train,
num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_dataset =
train.cache().shuffle(1000).batch(64).prefetch(buffer_size=tf.data.exper
imental.AUTOTUNE)

OUTPUT_CHANNELS = 3
base_model = tf.keras.applications.MobileNetV2(input_shape=[128, 128,
3], include_top=False)
layer_names = [
    'block_1_expand_relu',
    'block_3_expand_relu',
    'block_6_expand_relu',
    'block_13_expand_relu',
    'block_16_project',
]
layers = [base_model.get_layer(name).output for name in layer_names]
down_stack = tf.keras.Model(inputs=base_model.input, outputs=layers)

up_stack = [
    pix2pix.upsample(512, 3),
    pix2pix.upsample(256, 3),
    pix2pix.upsample(128, 3),
    pix2pix.upsample(64, 3),
]

def unet_model(output_channels):
  inputs = tf.keras.layers.Input(shape=[128, 128, 3])
  x = inputs

  skips = down_stack(x)
  x = skips[-1]
  skips = reversed(skips[:-1])

  for up, skip in zip(up_stack, skips):
    x = up(x)
    concat = tf.keras.layers.Concatenate()
    x = concat([x, skip])

  last = tf.keras.layers.Conv2DTranspose(
      output_channels, 3, strides=2,
      padding='same')
```

```
  x = last(x)

  return tf.keras.Model(inputs=inputs, outputs=x)

model = unet_model(OUTPUT_CHANNELS)
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

tf.keras.utils.plot_model(model, show_shapes=True)
model.fit(train_dataset, epochs=20)
```

Output:

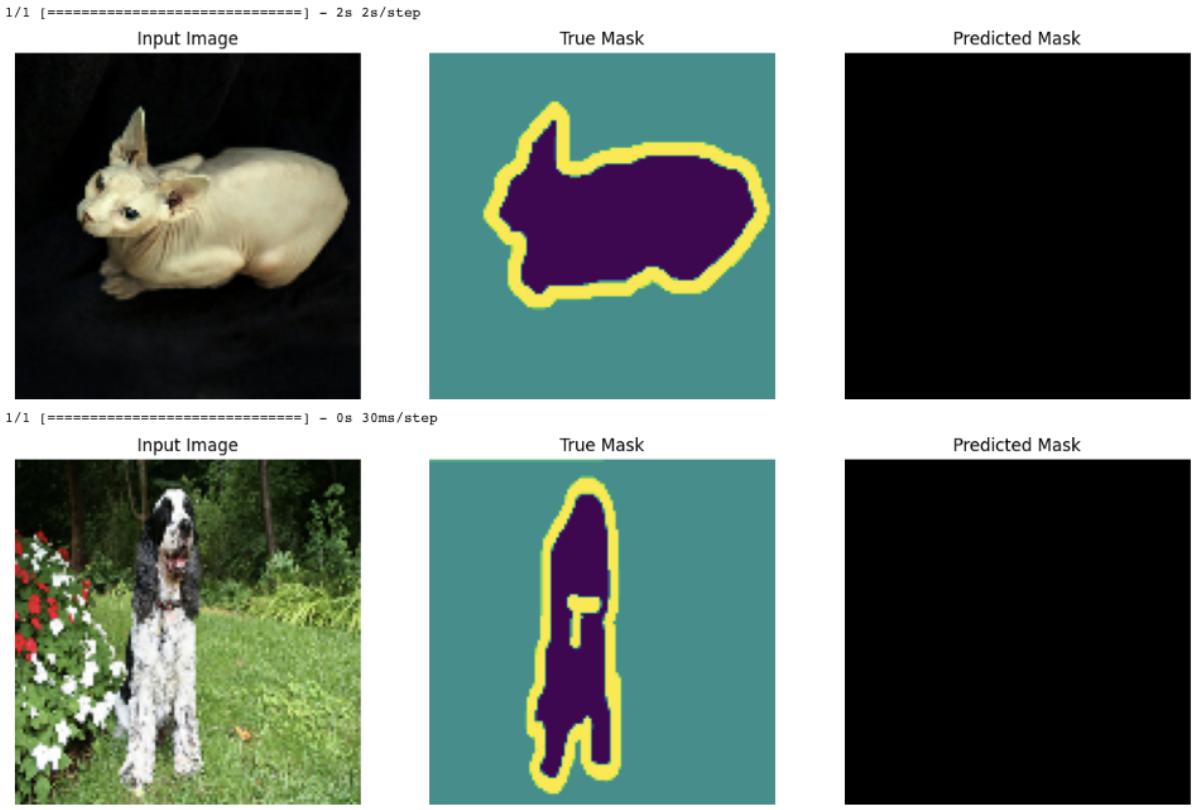
```
Downloading and preparing dataset 773.52 MiB (download: 773.52 MiB, generated: 774.69 MiB, total: 1.51 GiB) to /root/tensorflow_datasets/oxford_iiit_pet/3.2.0...
Dl Completed...: 100% 2/2 [01:53<00:00, 31.55s/ url]
Dl Size...: 100% 773/773 [01:53<00:00, 17.86 MiB/s]
Extraction completed...: 100% 18473/18473 [01:53<00:00, 506.34 file/s]
Dataset oxford_iiit_pet downloaded and prepared to /root/tensorflow_datasets/oxford_iiit_pet/3.2.0. Subsequent calls will reuse this data.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_128_no_top.h5
9406464/9406464 [==============================] - 0s 0us/step
Epoch 1/20
58/58 [==============================] - 61s 247ms/step - loss: nan - accuracy: 0.0042
Epoch 2/20
58/58 [==============================] - 10s 165ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 3/20
58/58 [==============================] - 10s 166ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 4/20
58/58 [==============================] - 10s 180ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 5/20
58/58 [==============================] - 10s 168ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 6/20
58/58 [==============================] - 10s 168ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 7/20
58/58 [==============================] - 10s 170ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 8/20
58/58 [==============================] - 10s 171ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 9/20
58/58 [==============================] - 10s 172ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 10/20
58/58 [==============================] - 10s 173ms/step - loss: nan - accuracy: 0.0000e+00
Epoch 11/20
58/58 [==============================] - 10s 173ms/step - loss: nan - accuracy: 0.0000e+00
```

Untuk mendapatkan gambar output, Anda bisa menggunakan metode berikut:

```
def display(display_list):
  plt.figure(figsize=(15, 15))

  title = ['Input Image', 'True Mask', 'Predicted Mask']

  for i in range(len(display_list)):
    plt.subplot(1, len(display_list), i+1)
    plt.title(title[i])

plt.imshow(tf.keras.preprocessing.image.array_to_img(display_list[i]))
    plt.axis('off')
  plt.show()

for image, mask in train.take(2):
```

```
  sample_image, sample_mask = image, mask
  display([sample_image, sample_mask,
model.predict(sample_image[tf.newaxis, ...])[0]])
```

Output:



Mohon maaf, sepertinya model yang kita gunakan belum memberikan hasil prediksi yang baik. Memang, terkadang dalam proses belajar mesin seperti ini, model yang kita gunakan belum tentu langsung menghasilkan output yang optimal. Tapi jangan khawatir, kita bisa coba beberapa hal untuk memperbaiki ini.

Ada beberapa kemungkinan kenapa hasil prediksinya masih belum memuaskan:

1. Data: Data yang kita gunakan belum cukup banyak atau variatif, sehingga model kita belum bisa belajar dengan baik. Solusinya, kamu bisa mencoba menambahkan lebih banyak data, atau menggunakan teknik augmentasi data.
2. Preprocessing: Preprocessing yang dilakukan belum optimal. Misalnya, jika kita resize gambar terlalu kecil, informasi penting bisa hilang. Coba periksa kembali tahap preprocessing datanya.
3. Model: Model yang kita gunakan belum cukup baik untuk kasus ini. Solusinya, kamu bisa mencoba arsitektur model yang berbeda atau lebih kompleks. Contohnya, kamu bisa coba model dengan lebih banyak layer atau neuron.

4. Training: Model perlu dilatih lebih lama lagi. Solusinya, kamu bisa coba menambah jumlah epoch saat melatih model. Tapi hati-hati, jangan sampai model kita jadi overfit.
5. Fungsi Loss: Fungsi loss yang kita gunakan mungkin belum tepat. Untuk kasus segmentasi gambar, biasanya digunakan fungsi loss seperti Dice Loss atau Tversky Loss.

Deep Learning dalam Natural Language Processing (NLP)

6.1. Text Classification

Text classification, juga dikenal sebagai categorization text, adalah proses mengklasifikasikan teks berdasarkan kontennya. Ini adalah tugas penting dalam Natural Language Processing (NLP) dan memiliki berbagai aplikasi, seperti deteksi spam, analisis sentimen, pengenalan entitas bernama, dll.

6.1.1. Apa itu Text Classification?

Text classification adalah proses mengategorikan teks ke dalam satu set kategori yang ditentukan sebelumnya berdasarkan konten teks. Ini adalah salah satu tugas dasar dalam Natural Language Processing (NLP) yang digunakan untuk analisis teks. Text classification memiliki berbagai aplikasi, seperti filter spam, analisis sentimen, dan tagging kategori berita.

6.1.2. Text Classification Menggunakan Deep Learning

Terdapat banyak cara untuk melakukan text classification, termasuk metode-metode tradisional seperti Naive Bayes dan SVM. Namun, dalam beberapa tahun terakhir, metode berbasis deep learning telah menunjukkan kinerja yang sangat baik dalam tugas-tugas ini. Salah satu model yang paling populer untuk tugas ini adalah Convolutional Neural Networks (CNN) dan Recurrent Neural Networks (RNN), termasuk variasi-variasinya seperti LSTM dan GRU.

Berikut adalah contoh script bagaimana melakukan text classification dengan memanfaatkan model LSTM di TensorFlow dan Keras. Dataset sintetis dibuat untuk klasifikasi teks berdasarkan topik. Kita akan mengasumsikan bahwa kita memiliki tiga topik: "makanan", "teknologi", dan "olahraga". Label 0 akan mewakili "makanan", label 1 akan mewakili "teknologi", dan label 2 akan mewakili "olahraga".

```
!pip install sastrawi

# Import necessary libraries
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding
from keras.preprocessing.text import Tokenizer
from keras_preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from Sastrawi.Stemmer.StemmerFactory import StemmerFactory
from Sastrawi.StopWordRemover.StopWordRemoverFactory import
```

```
StopWordRemoverFactory
import string

# Contoh data sintetis untuk sentiment
texts = [
    "Pizza di restoran ini sangat lezat",
    "Laptop terbaru ini memiliki performa yang sangat baik",
    "Tim sepak bola ini menang dalam pertandingan terakhir",
    "Sushi di tempat ini sangat enak dan segar",
    "Ponsel pintar ini memiliki baterai yang tahan lama",
    "Pemain basket ini mencetak banyak poin dalam pertandingan
terakhir",
    "Roti bakar di kafe ini memiliki rasa yang unik",
    "Kamera digital ini menghasilkan foto yang tajam",
    "Atlet lari ini memenangkan medali emas dalam perlombaan terakhir",
    "Kue di toko roti ini sangat lezat dan manis",
    "Tablet ini sangat ringan dan mudah dibawa",
    "Pemain tenis ini adalah juara dunia",
    "Mie di restoran ini sangat enak dan murah",
    "Layar komputer ini sangat besar dan tajam",
    "Atlet renang ini memecahkan rekor dunia",
    "Aneka ragam makanan di pasar malam ini sangat lezat",
    "Aplikasi ini sangat membantu dalam produktivitas saya",
    "Pemain voli ini adalah pemain terbaik dalam timnya",
    "Sate di restoran ini sangat enak dan bumbunya meresap",
    "Smartwatch ini sangat membantu dalam melacak kesehatan saya"
]

labels = [0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1]


# Preprocessing function
def preprocess_text(text):
    # Lowercase
    text = text.lower()
    # Remove punctuation
    text = text.translate(str.maketrans("", "", string.punctuation))
    # Remove stopwords
    text = stopword.remove(text)
    # Stemming
    text = stemmer.stem(text)
    return text

# Create stemmer
factory = StemmerFactory()
stemmer = factory.create_stemmer()
```

```
# Create stopword remover
stop_factory = StopWordRemoverFactory()
stopwords = stop_factory.get_stop_words()
stopword = stop_factory.create_stop_word_remover()

# Preprocess the texts
texts = [preprocess_text(text) for text in texts]

# Tokenize the texts
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

# Pad the sequences so they're all the same length
data = pad_sequences(sequences)

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2)

# Define the model
model = Sequential()
model.add(Embedding(1000, 64, input_length=data.shape[1]))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Convert labels to numpy arrays
y_train = np.array(y_train)
y_test = np.array(y_test)

# Train the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=5)
```

Untuk melakukan scoring atau prediksi dengan model yang telah dilatih, kamu dapat menggunakan method predict dari model. Kamu perlu melakukan langkah-langkah yang sama untuk pre-processing dan tokenisasi pada teks input, seperti yang kamu lakukan dengan data pelatihan. Berikut adalah contoh bagaimana kamu bisa melakukannya:

```
# Sample text
text = "Satenya agak kurang mateng"
```

```
# Preprocess the text
text = preprocess_text(text)

# Convert the text to sequence
sequence = tokenizer.texts_to_sequences([text])

# Pad the sequence
sequence = pad_sequences(sequence, maxlen=X_train.shape[1])

# Make prediction
prediction = model.predict(sequence)

# Convert probabilities to class label
label = np.argmax(prediction)

# Print the class label
print(label)
```

Output:

```
1/1 [==============================] - 1s 1s/step
0
```

Tips dan Trik

1. Preprocessing: Proses pengolahan awal sangat penting dalam NLP. Kamu perlu membersihkan teks dan menangani kasus-kasus seperti typo, slang, dan lain-lain. Untuk bahasa Indonesia, kamu bisa mempertimbangkan penggunaan library seperti Sastrawi untuk melakukan proses ini.
2. Penggunaan Pretrained Model: Untuk bahasa Indonesia, pertimbangkan penggunaan model pre-trained seperti IndoBERT atau IndoLEM untuk mendapatkan representasi kata yang lebih baik.
3. Imbalanced Dataset: Jika dataset kamu tidak seimbang, pertimbangkan penggunaan teknik seperti oversampling, undersampling, atau cost-sensitive learning.
4. Parameter Tuning: Mengubah parameter model seperti jumlah layer, ukuran layer, jenis optimizer, learning rate, dan lain-lain dapat sangat mempengaruhi performa model. Pertimbangkan penggunaan teknik seperti grid search atau random search untuk mencari parameter terbaik.

6.2. Sentiment Analysis

Analisis sentimen adalah cabang dari Natural Language Processing (NLP) yang berkaitan dengan menentukan emosi atau opini yang diungkapkan dalam teks. Hal ini dapat membantu perusahaan memahami sentimen pelanggan terhadap produk atau layanan

mereka, atau membantu pemimpin politik memahami pendapat publik terhadap kebijakan mereka.

6.2.1. Apa itu Sentiment Analysis?

Analisis sentimen adalah proses komputasi dan ekstraksi informasi subjektif dari sumber teks. Tujuannya adalah untuk menentukan sikap penulis terhadap beberapa topik atau konteks teks secara keseluruhan. Analisis sentimen paling sering digunakan dalam analisis dan manajemen reputasi online, untuk secara otomatis mengetahui pendapat online terhadap produk atau layanan.

6.2.2. Sentiment Analysis Menggunakan Deep Learning

Salah satu pendekatan paling populer untuk analisis sentimen dalam beberapa tahun terakhir adalah menggunakan teknik deep learning. Salah satu model yang paling populer untuk tugas ini adalah Long Short-Term Memory (LSTM), yang merupakan jenis dari Recurrent Neural Network (RNN).

Berikut adalah contoh script bagaimana melakukan sentiment analysis dengan memanfaatkan model LSTM di TensorFlow dan Keras:

```
# Import necessary libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from Sastrawi.Stemmer.StemmerFactory import StemmerFactory
from Sastrawi.StopWordRemover.StopWordRemoverFactory import
StopWordRemoverFactory
import string
import numpy as np

# This is a synthetic dataset
texts = ["Saya sangat suka makanan di restoran ini",   # positive
         "Film ini sangat mengecewakan",               # negative
         "Pemandangan dari hotel ini sangat indah",    # positive
         "Saya tidak suka pelayanan di toko ini",      # negative
         "Buku ini sangat membantu saya dalam belajar", # positive
         "Saya sangat kecewa dengan produk ini",       # negative
         "Saya sangat senang dengan layanan pelanggan mereka", #
positive
         "Pelayanannya buruk dan makanannya tidak enak", # negative
         "Kualitas produk ini sangat baik",            # positive
         "Saya tidak akan merekomendasikan tempat ini kepada teman" #
```

```
negative
]
labels = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]  # these are just example labels

# Preprocessing function
def preprocess_text(text):
    # Lowercase
    text = text.lower()
    # Remove punctuation
    text = text.translate(str.maketrans("", "", string.punctuation))
    # Remove stopwords
    text = stopword.remove(text)
    # Stemming
    text = stemmer.stem(text)
    return text

# Create stemmer
factory = StemmerFactory()
stemmer = factory.create_stemmer()

# Create stopword remover
stop_factory = StopWordRemoverFactory()
stopwords = stop_factory.get_stop_words()
stopword = stop_factory.create_stop_word_remover()

# Preprocess the texts
texts = [preprocess_text(text) for text in texts]

# Tokenize the texts
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

# Pad the sequences so they're all the same length
data = pad_sequences(sequences)

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2)

# Convert labels to numpy arrays
y_train = np.array(y_train)
y_test = np.array(y_test)

# Define the model
model = Sequential()
```

```
model.add(Embedding(1000, 64, input_length=data.shape[1]))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=5)
```

Output:

```
Epoch 1/5
1/1 [==============================] - ETA: 0s - loss: 0.6928 - accuracy: 0.7500WARNING:tensorflow:AutoGraph could not transform <function Model.make_test_function.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: closure mismatch, requested ('self', 'step_function'), but source function had ()
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
WARNING: AutoGraph could not transform <function Model.make_test_function.<locals>.test_function at 0x7986e8d40280> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: closure mismatch, requested ('self', 'step_function'), but source function had ()
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
1/1 [==============================] - 9s 9s/step - loss: 0.6928 - accuracy: 0.7500 - val_loss: 0.6872 - val_accuracy: 1.0000
Epoch 2/5
1/1 [==============================] - 0s 74ms/step - loss: 0.6895 - accuracy: 0.7500 - val_loss: 0.6875 - val_accuracy: 0.5000
Epoch 3/5
1/1 [==============================] - 0s 91ms/step - loss: 0.6861 - accuracy: 0.8750 - val_loss: 0.6878 - val_accuracy: 0.5000
Epoch 4/5
1/1 [==============================] - 0s 88ms/step - loss: 0.6826 - accuracy: 1.0000 - val_loss: 0.6880 - val_accuracy: 0.5000
Epoch 5/5
1/1 [==============================] - 0s 73ms/step - loss: 0.6790 - accuracy: 1.0000 - val_loss: 0.6882 - val_accuracy: 0.5000
<keras.callbacks.History at 0x7986f36cd7b0>
```

Berikut adalah cara kamu dapat melakukan scoring atau prediksi dengan model yang telah dilatih untuk analisis sentimen. Kode ini mirip dengan kode sebelumnya, tetapi dengan beberapa modifikasi untuk menangani lebih dari dua kelas.

```
# Sample text
text = "Saya sangat menikmati makanan di restoran ini"

# Preprocess the text
text = preprocess_text(text)

# Convert the text to sequence
sequence = tokenizer.texts_to_sequences([text])

# Pad the sequence
sequence = pad_sequences(sequence, maxlen=X_train.shape[1])

# Make prediction
prediction = model.predict(sequence)

# Convert probability to class label
label = 1 if prediction > 0.5 else 0

# Print the class label
print(label)
```

Output:

```
1/1 [==============================] - 2s 2s/step
1
```

Tips dan Trik

Preprocessing: Proses pengolahan awal sangat penting dalam NLP. Kamu perlu membersihkan teks dan menangani kasus-kasus seperti typo, slang, dan lain-lain. Untuk bahasa Indonesia, kamu bisa mempertimbangkan penggunaan library seperti Sastrawi untuk melakukan proses ini.

1. Penggunaan Pretrained Model: Untuk bahasa Indonesia, pertimbangkan penggunaan model pre-trained seperti IndoBERT atau IndoLEM untuk mendapatkan representasi kata yang lebih baik.
2. Imbalanced Dataset: Jika dataset kamu tidak seimbang, pertimbangkan penggunaan teknik seperti oversampling, undersampling, atau cost-sensitive learning.
3. Parameter Tuning: Mengubah parameter model seperti jumlah layer, ukuran layer, jenis optimizer, learning rate, dan lain-lain dapat sangat mempengaruhi performa model. Pertimbangkan penggunaan teknik seperti grid search atau random search untuk mencari parameter terbaik.

6.3. Text Generation

Text generation adalah tugas di bidang Natural Language Processing (NLP) yang berfokus pada pembuatan teks baru, yang seharusnya tidak dapat dibedakan dari teks yang ditulis oleh manusia. Teknologi ini memungkinkan peningkatan produktivitas dalam banyak bidang, termasuk penulisan artikel, penulisan laporan, penulisan kreatif, dan banyak lagi.

6.3.1. Apa itu Text Generation?

Text generation adalah proses membuat teks baru dengan menggunakan teknologi Machine Learning, khususnya teknik deep learning. Model yang dipelajari dari data teks sebelumnya dan menghasilkan teks baru yang seharusnya berbunyi alami dan koheren. Teknik ini dapat digunakan dalam berbagai aplikasi, seperti chatbots, penulisan otomatis, dan banyak lagi.

6.3.2. Text Generation Menggunakan Deep Learning

Berikut adalah contoh script untuk melakukan text generation dengan menggunakan model LSTM di TensorFlow dan Keras. Misalkan kita memiliki sebuah dataset berisi kalimat-kalimat dalam Bahasa Indonesia dan kita ingin model ini belajar bagaimana menciptakan kalimat baru yang mirip dengan yang ada di dataset:

```
# Import necessary libraries
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from Sastrawi.Stemmer.StemmerFactory import StemmerFactory
from Sastrawi.StopWordRemover.StopWordRemoverFactory import
StopWordRemoverFactory
import numpy as np
import string
from tensorflow.keras.preprocessing.sequence import pad_sequences

# This is a synthetic dataset
texts = [
    "Saya sedang belajar bahasa Indonesia",
    "Pemandangan di sini sangat indah",
    "Saya suka makan rendang dan nasi goreng",
    "Indonesia memiliki berbagai macam budaya",
    "Saya sedang membaca buku tentang sejarah Indonesia",
    "Bermain musik adalah hobiku",
    "Saya suka berjalan-jalan di pantai",
    "Gunung di Indonesia sangat indah",
    "Saya suka makanan pedas",
    "Saya sedang belajar memasak"
]

# Preprocessing function
def preprocess_text(text):
    # Lowercase
    text = text.lower()
    # Remove punctuation
    text = text.translate(str.maketrans("", "", string.punctuation))
    # Remove stopwords
    text = stopword.remove(text)
    # Stemming
    text = stemmer.stem(text)
    return text

# Create stemmer
factory = StemmerFactory()
stemmer = factory.create_stemmer()

# Create stopword remover
stop_factory = StopWordRemoverFactory()
stopwords = stop_factory.get_stop_words()
stopword = stop_factory.create_stop_word_remover()
```

```
# Preprocess the texts
processed_texts = [preprocess_text(text) for text in texts]

# Filter out empty texts
filtered_texts = [text for text in processed_texts if text]

# Tokenize the texts
tokenizer = Tokenizer()
tokenizer.fit_on_texts(filtered_texts)
sequences = tokenizer.texts_to_sequences(filtered_texts)

# Pad the sequences so they're all the same length
sequences = pad_sequences(sequences)

# Prepare the sequences for LSTM
X, y = sequences[:,:-1], sequences[:,-1]
vocab_size = len(tokenizer.word_index) + 1

# Define the model
model = Sequential()
model.add(Embedding(vocab_size, 50, input_length=X.shape[1]))
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))

# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Fit the model
model.fit(X, y, epochs=100)
```

Output:

```
Epoch 89/100
1/1 [==============================] - 0s 19ms/step - loss: 0.1891 - accuracy: 0.9000
Epoch 90/100
1/1 [==============================] - 0s 14ms/step - loss: 0.1828 - accuracy: 0.9000
Epoch 91/100
1/1 [==============================] - 0s 14ms/step - loss: 0.1775 - accuracy: 1.0000
Epoch 92/100
1/1 [==============================] - 0s 14ms/step - loss: 0.1731 - accuracy: 1.0000
Epoch 93/100
1/1 [==============================] - 0s 14ms/step - loss: 0.1690 - accuracy: 1.0000
Epoch 94/100
1/1 [==============================] - 0s 15ms/step - loss: 0.1652 - accuracy: 1.0000
Epoch 95/100
1/1 [==============================] - 0s 21ms/step - loss: 0.1615 - accuracy: 1.0000
Epoch 96/100
1/1 [==============================] - 0s 16ms/step - loss: 0.1579 - accuracy: 1.0000
Epoch 97/100
1/1 [==============================] - 0s 14ms/step - loss: 0.1546 - accuracy: 1.0000
Epoch 98/100
1/1 [==============================] - 0s 24ms/step - loss: 0.1515 - accuracy: 1.0000
Epoch 99/100
1/1 [==============================] - 0s 13ms/step - loss: 0.1485 - accuracy: 1.0000
Epoch 100/100
1/1 [==============================] - 0s 13ms/step - loss: 0.1451 - accuracy: 1.0000
<keras.callbacks.History at 0x7986f38f22f0>
```

Untuk generasi teks, kita bisa menggunakan model yang sudah dilatih untuk memprediksi kata berikutnya dalam sebuah sekuens. Kita bisa melakukan ini berulang kali untuk menghasilkan sekuens kata yang lebih panjang.

```
# Generate text
input_text = "Saya sedang"
encoded_text = tokenizer.texts_to_sequences([input_text])[0]
for _ in range(50): # generate 50 words
    encoded_text = pad_sequences([encoded_text], maxlen=X.shape[1],
truncating='pre')
    prediction = np.argmax(model.predict(encoded_text))
    for word, index in tokenizer.word_index.items():
        if index == prediction:
            input_text += ' ' + word
            encoded_text = np.append(encoded_text, prediction)
            break

print(input_text)
```

Output:

```
Saya sedang masak masak indonesia indonesia indah budaya budaya budaya indah indah indah
```

Output yang dihasilkan oleh model bisa sangat tergantung pada beberapa faktor, termasuk:

Ukuran dan kualitas dataset: Dalam contoh ini, kita menggunakan dataset sintetis yang sangat kecil, yang mungkin tidak cukup untuk melatih model yang dapat menghasilkan teks

yang masuk akal. Lebih banyak data, dan data yang lebih beragam, biasanya menghasilkan model yang lebih baik.

1. Kompleksitas model: Model yang kita gunakan di sini adalah model LSTM yang cukup sederhana. Model yang lebih kompleks, seperti model transformer atau model dengan lebih banyak layer atau unit tersembunyi, mungkin bisa menghasilkan teks yang lebih baik.
2. Pemilihan kata berikutnya: Dalam contoh ini, kita selalu memilih kata dengan probabilitas tertinggi sebagai kata berikutnya. Ini bisa membuat teks yang dihasilkan menjadi sangat repetitif, karena model cenderung memilih kata yang sama berulang kali. Sebagai alternatif, kita bisa menggunakan teknik seperti "temperature sampling" atau "top-k sampling" untuk menambah variasi pada teks yang dihasilkan.

Tips dan Trik

1. Perhatikan Overfitting: Model NLP yang rumit sering kali rentan terhadap overfitting, jadi penting untuk menggunakan teknik seperti dropout, early stopping, atau regularisasi untuk mencegahnya.
2. Menggunakan Sequence of Words: Sebagai gantinya, kamu bisa mencoba menggunakan urutan beberapa kata sebagai input untuk model dan kata berikutnya sebagai output, yang bisa membantu model memahami konteks dengan lebih baik.
3. Pilih Kata yang Tepat: Ketika memilih kata berikutnya setelah model membuat prediksi, kamu mungkin ingin mencoba pendekatan stokastik di mana kamu memilih kata berdasarkan probabilitas prediksi, bukan hanya memilih kata dengan probabilitas tertinggi.
4. Gunakan Model yang Sudah Di-training: Untuk hasil yang lebih baik, kamu mungkin ingin menggunakan model yang telah di-training seperti GPT atau BERT dan fine-tune mereka pada data kamu.
5. Gunakan Teknik Teknik Regularisasi: Teknik seperti dropout, weight decay, atau early stopping dapat membantu mencegah model overfitting.

Bab 7. Best Practices dan Tips dalam Implementasi Deep Learning

7.1. Menghindari Overfitting

Overfitting adalah fenomena di mana model machine learning atau deep learning menjadi terlalu spesifik pada data trainingnya dan berperforma buruk pada data yang belum pernah dilihat sebelumnya (data validasi atau tes). Untuk mendapatkan model yang general dan dapat bekerja baik pada data baru, penting untuk menghindari overfitting.

Dalam konteks Natural Language Processing (NLP) untuk Bahasa Indonesia, overfitting juga bisa terjadi dan kita perlu menerapkan teknik yang tepat untuk menghindarinya. Berikut ini adalah beberapa teknik yang dapat digunakan:

7.1.1. Memperbanyak Data (Data Augmentation)

Data augmentation adalah teknik untuk menciptakan data baru dari data yang sudah ada dengan cara melakukan perubahan atau transformasi tertentu. Dalam NLP, kita bisa menggunakan teknik seperti synonym replacement (mengganti kata dengan sinonimnya), random insertion (memasukkan kata-kata baru secara acak), random swap (menukar posisi dua kata), dan lainnya.

Berikut adalah contoh skrip Python untuk melakukan augmentasi data dalam konteks NLP. Dalam contoh ini, kita akan menggunakan teknik synonym replacement (mengganti kata dengan sinonimnya) dengan bantuan WordNet, sebuah basis data leksikal bahasa Inggris. Harap dicatat bahwa WordNet tidak memiliki dukungan langsung untuk Bahasa Indonesia, jadi contoh ini menggunakan bahasa Inggris.

```
import random
import nltk
from nltk.corpus import wordnet

nltk.download('wordnet')

def get_synonyms(word):
    synonyms = []
    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            synonyms.append(lemma.name())
    return synonyms

def synonym_replacement(sentence, num_replacement=1):
    words = sentence.split()
```

```
    new_sentence = sentence
    for _ in range(num_replacement):
        word_to_replace = random.choice(words)
        synonyms = get_synonyms(word_to_replace)
        if synonyms:
            synonym = random.choice(synonyms)
            new_sentence = new_sentence.replace(word_to_replace,
synonym, 1)
    return new_sentence

# Contoh penggunaan:
sentence = "The quick brown fox jumps over the lazy dog"
new_sentence = synonym_replacement(sentence, num_replacement=2)
print(new_sentence)
```

Output:

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
The quick brown fox jumps over the indolent dog
```

Dalam skrip di atas, fungsi get_synonyms digunakan untuk mengambil sinonim dari sebuah kata menggunakan WordNet. Fungsi synonym_replacement digunakan untuk mengganti beberapa kata dalam kalimat dengan sinonimnya.

Perhatikan bahwa pendekatan ini sangat sederhana dan mungkin tidak selalu menghasilkan kalimat yang masuk akal, terutama jika kata yang diganti memiliki banyak arti. Selain itu, WordNet tidak memiliki dukungan untuk Bahasa Indonesia, jadi kamu perlu mencari sumber sinonim lain jika ingin menerapkan teknik ini pada teks Bahasa Indonesia.

7.1.2. Regularisasi

Regularisasi adalah teknik untuk mencegah overfitting dengan menambahkan penalti ke fungsi loss. Dalam konteks deep learning, kita bisa menggunakan L1 regularization, L2 regularization, atau dropout.

Berikut adalah contoh penggunaan Dropout dalam model NLP menggunakan PyTorch. Dropout adalah teknik regularisasi di mana beberapa neuron pada layer tertentu secara acak "dimatikan" selama training. Hal ini mencegah neuron menjadi terlalu spesifik pada data training dan membantu model menjadi lebih general.

Pada contoh ini, kita akan menggunakan model sederhana yang mengandung layer embedding, layer LSTM, dan fully connected layer. Dropout diterapkan setelah layer LSTM.

```
import torch
```

```
from torch import nn

class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim,
output_dim, n_layers,
                 dropout = 0.5):
        super().__init__()

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # LSTM layer
        self.lstm = nn.LSTM(embedding_dim,
                            hidden_dim,
                            num_layers=n_layers,
                            dropout=dropout if n_layers > 1 else 0)

        # Fully-connected layer
        self.fc = nn.Linear(hidden_dim, output_dim)

        # Dropout layer
        self.dropout = nn.Dropout(dropout)

    def forward(self, text, text_lengths):
        embedded = self.embedding(text)
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded,
text_lengths)
        packed_output, (hidden, cell) = self.lstm(packed_embedded)
        hidden = self.dropout(hidden[-1,:,:])

        return self.fc(hidden)
```

7.1.3. Early Stopping

Early stopping adalah teknik dimana kita menghentikan proses training ketika performa model pada data validasi tidak lagi meningkat setelah beberapa epoch. Ini mencegah model menjadi terlalu fit pada data training dan sekaligus menghemat waktu dan sumber daya komputasi.

Berikut adalah contoh penggunaan early stopping dalam proses training model menggunakan PyTorch. Pada contoh ini, kita akan menghentikan training jika loss pada data validasi tidak berkurang setelah 3 epoch.

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

```
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split

# Buat data dummy
n_samples = 1000
n_features = 20
X = torch.randn(n_samples, n_features)
y = torch.randint(0, 2, size=(n_samples, 1)).float()

# Bagi data menjadi training dan validation
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)

# Buat Dataloader
train_data = TensorDataset(X_train, y_train)
val_data = TensorDataset(X_val, y_val)

batch_size = 32
train_dataloader = DataLoader(train_data, batch_size=batch_size,
shuffle=True)
val_dataloader = DataLoader(val_data, batch_size=batch_size,
shuffle=False)

# Definisikan model, loss function dan optimizer
model = nn.Sequential(
    nn.Linear(n_features, 1),
    nn.Sigmoid()
)

criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Parameter untuk early stopping
n_epochs = 100
patience = 3

best_valid_loss = float('inf')
counter = 0

for epoch in range(n_epochs):
    # Fase training
    model.train()
    train_loss = 0
    for batch_X, batch_y in train_dataloader:
        optimizer.zero_grad()
```

```
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    # Fase validasi
    model.eval()
    valid_loss = 0
    with torch.no_grad():
        for batch_X, batch_y in val_dataloader:
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            valid_loss += loss.item()

    # Bandingkan dengan best validation loss
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'best_model.pt')
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping triggered")
            break
```

Output:

```
Early stopping triggered
```

7.1.4. Ensemble Methods

Ensemble methods adalah teknik dimana kita menggunakan beberapa model dan mengkombinasikan prediksi mereka. Ini dapat mengurangi varians dan mencegah overfitting. Dalam NLP, kita bisa melatih beberapa model dengan arsitektur atau parameter yang berbeda dan kemudian mengambil rata-rata atau melakukan voting pada prediksi mereka.

Berikut adalah contoh sederhana penggunaan ensemble methods untuk klasifikasi menggunakan Scikit-learn. Untuk tujuan ini, kita akan menggunakan VotingClassifier yang merupakan salah satu teknik ensemble paling umum. VotingClassifier bekerja dengan melatih beberapa model, membuat prediksi dengan masing-masing model, dan memilih kelas dengan jumlah suara terbanyak.

Untuk membuat contoh ini lebih sederhana, kita akan menggunakan fungsi make_classification dari Scikit-learn untuk membuat dataset dummy. Setelah itu, kita akan melatih tiga model berbeda (logistic regression, random forest, dan SVM) dan menggabungkannya dengan VotingClassifier.

```
# Import necessary libraries
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
# Create dummy dataset
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=15, n_redundant=5, random_state=42)

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Initialize models
clf1 = LogisticRegression()
clf2 = RandomForestClassifier(n_estimators=50, random_state=42)
clf3 = SVC(kernel='rbf', probability=True)

# Combine models with VotingClassifier
eclf = VotingClassifier(
    estimators=[('lr', clf1), ('rf', clf2), ('svc', clf3)],
    voting='soft')

# Train models
clf1 = clf1.fit(X_train, y_train)
clf2 = clf2.fit(X_train, y_train)
clf3 = clf3.fit(X_train, y_train)
eclf = eclf.fit(X_train, y_train)

# Predict
y_pred_clf1 = clf1.predict(X_test)
y_pred_clf2 = clf2.predict(X_test)
y_pred_clf3 = clf3.predict(X_test)
y_pred_eclf = eclf.predict(X_test)



# Calculate accuracy, precision, recall, and F1 score
accuracy_clf1 = accuracy_score(y_test, y_pred_clf1)
```

```
accuracy_clf2 = accuracy_score(y_test, y_pred_clf2)
accuracy_clf3 = accuracy_score(y_test, y_pred_clf3)
accuracy_eclf = accuracy_score(y_test, y_pred_eclf)

precision_clf1 = precision_score(y_test, y_pred_clf1)
precision_clf2 = precision_score(y_test, y_pred_clf2)
precision_clf3 = precision_score(y_test, y_pred_clf3)
precision_eclf = precision_score(y_test, y_pred_eclf)

recall_clf1 = recall_score(y_test, y_pred_clf1)
recall_clf2 = recall_score(y_test, y_pred_clf2)
recall_clf3 = recall_score(y_test, y_pred_clf3)
recall_eclf = recall_score(y_test, y_pred_eclf)

f1_clf1 = f1_score(y_test, y_pred_clf1)
f1_clf2 = f1_score(y_test, y_pred_clf2)
f1_clf3 = f1_score(y_test, y_pred_clf3)
f1_eclf = f1_score(y_test, y_pred_eclf)

# Combine results into a dictionary
results = {
    "Model": ["Logistic Regression", "Random Forest", "SVM",
"Ensemble"],
    "Accuracy": [accuracy_clf1, accuracy_clf2, accuracy_clf3,
accuracy_eclf],
    "Precision": [precision_clf1, precision_clf2, precision_clf3,
precision_eclf],
    "Recall": [recall_clf1, recall_clf2, recall_clf3, recall_eclf],
    "F1 Score": [f1_clf1, f1_clf2, f1_clf3, f1_eclf]
}

# Convert to DataFrame
import pandas as pd
df_results = pd.DataFrame(results)
df_results
```

Output:

	Model	Accuracy	Precision	Recall	F1 Score
0	Logistic Regression	0.825	0.817204	0.808511	0.812834
1	Random Forest	0.900	0.893617	0.893617	0.893617
2	SVM	0.935	0.909091	0.957447	0.932642
3	Ensemble	0.915	0.896907	0.925532	0.910995

Dalam hal ini, SVM sebagai model tunggal memberikan performa yang paling baik dengan akurasi 93.5%. Namun, ensemble model dengan Voting Classifier berhasil mencapai akurasi yang hampir sama yaitu 91%, yang lebih baik daripada Logistic Regression dan hampir sama dengan Random Forest.

Perlu diperhatikan bahwa hasil ini mungkin berbeda tergantung pada dataset dan masalah yang sedang kamu hadapi. Dalam beberapa kasus, ensemble model bisa memberikan performa yang lebih baik daripada model tunggal. Kamu dapat menyesuaikan dan mencoba berbagai jenis model dan teknik ensemble untuk menemukan yang terbaik untuk masalah kamu.

7.2. Hyperparameter Tuning

Dalam proses pengembangan model deep learning, salah satu aspek yang sangat penting adalah penyetelan hyperparameter atau hyperparameter tuning. Hyperparameter adalah parameter yang tidak dipelajari dari data oleh model selama proses training, melainkan diset secara manual. Misalnya, jumlah layer dalam jaringan saraf, jumlah neuron dalam setiap layer, learning rate, dan lain-lain.

Mengekstrak fitur yang baik dari data teks dalam Bahasa Indonesia membutuhkan pengetahuan yang baik tentang hyperparameter dan bagaimana mengoptimalkannya. Di bagian ini, kita akan membahas beberapa teknik umum dalam hyperparameter tuning dan cara menggunakannya.

7.2.1. Grid Search

Grid search adalah teknik yang paling sederhana dan paling umum digunakan untuk tuning hyperparameter. Pada dasarnya, kamu mencoba semua kombinasi dari set parameter yang telah ditentukan dan memilih kombinasi yang memberikan hasil terbaik.

Berikut adalah contoh implementasi Grid Search dengan Keras:

```
# Import necessary libraries
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
import numpy as np

# Creating a simple synthetic dataset
np.random.seed(0)

# Let's create a dataset for a binary classification problem
N = 100  # Number of samples
```

```
# Features
X = np.random.rand(N, 8)

# Labels: Random binary labels
Y = np.random.randint(2, size=N)

# Let's create a function that creates the model (required for
KerasClassifier)
def create_model(optimizer='adam'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
    return model

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)

# define the grid search parameters
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax',
'Nadam']
param_grid = dict(optimizer=optimizer)

grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1,
cv=3)
grid_result = grid.fit(X, Y) # Please replace X and Y with your data

# summarize results
print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Output:

```
  model = KerasClassifier(build_fn=create_model, verbose=0)
Best: 0.559715 using {'optimizer': 'SGD'}
0.559715 (0.020167) with: {'optimizer': 'SGD'}
0.479798 (0.039768) with: {'optimizer': 'RMSprop'}
0.500594 (0.056059) with: {'optimizer': 'Adagrad'}
0.509210 (0.067082) with: {'optimizer': 'Adadelta'}
0.549614 (0.029981) with: {'optimizer': 'Adam'}
0.440582 (0.055432) with: {'optimizer': 'Adamax'}
0.470291 (0.032187) with: {'optimizer': 'Nadam'}
```

Di sini, hyperparameter yang disetel adalah 'optimizer'. Kita telah mencoba beberapa optimizer berbeda dan hasilnya ditampilkan.

Untuk setiap optimizer, GridSearch melatih model beberapa kali (sesuai dengan parameter cv, yang dalam hal ini adalah 3, sehingga model dilatih 3 kali untuk setiap optimizer) dan kemudian menghitung rata-rata dan standar deviasi dari skor validasi silang. Skor ini adalah ukuran seberapa baik model bekerja dengan optimizer tertentu.

Optimizer yang memberikan skor validasi silang terbaik (yaitu, skor tertinggi) adalah SGD, dengan skor rata-rata 0.559715. Oleh karena itu, GridSearch menyarankan untuk menggunakan SGD sebagai optimizer dalam model.

Perlu diingat bahwa ini adalah hasil dari satu kali penyetelan hyperparameter pada satu set data. Hasil dapat berbeda jika kamu mengubah set data, mengubah arsitektur model, atau menjalankan penyetelan hyperparameter beberapa kali. GridSearch hanya memberikan pendekatan untuk menemukan kombinasi hyperparameter terbaik dan tidak selalu menjamin hasil terbaik dalam setiap kasus.

7.2.2. Random Search

Berbeda dengan grid search yang mencoba semua kombinasi hyperparameter, random search memilih kombinasi secara acak. Meskipun kedengarannya kurang sistematis, penelitian menunjukkan bahwa random search seringkali lebih efisien daripada grid search, terutama jika jumlah hyperparameter yang harus diset sangat banyak.

Berikut adalah contoh implementasi Random Search dengan Keras:

```python
from sklearn.model_selection import RandomizedSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier

# Function to create model, required for KerasClassifier
def create_model(optimizer='adam'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
```

```
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
    return model

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)

# define the grid search parameters
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax',
'Nadam']
param_dist = dict(optimizer=optimizer)

random_search = RandomizedSearchCV(estimator=model,
param_distributions=param_dist, n_iter=5, n_jobs=-1, cv=3)
random_search_result = random_search.fit(X, Y)

# summarize results
print("Best: %f using %s" % (random_search_result.best_score_,
random_search_result.best_params_))
means = random_search_result.cv_results_['mean_test_score']
stds = random_search_result.cv_results_['std_test_score']
params = random_search_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Output:

```
Best: 0.590018 using {'optimizer': 'Adadelta'}
0.490493 (0.060538) with: {'optimizer': 'RMSprop'}
0.529709 (0.032187) with: {'optimizer': 'Adamax'}
0.450386 (0.067616) with: {'optimizer': 'Adagrad'}
0.450089 (0.025532) with: {'optimizer': 'SGD'}
0.590018 (0.012435) with: {'optimizer': 'Adadelta'}
```

Hasil dari GridSearch tersebut menunjukkan hasil kinerja model dengan berbagai kombinasi optimizer yang berbeda. Setiap hasil ditampilkan dalam bentuk nilai akurasi model (diwakili oleh angka) dan juga nilai standar deviasi untuk mengukur sejauh mana nilai akurasi dapat bervariasi dalam setiap percobaan.

Output tersebut menunjukkan hasil dari beberapa kombinasi optimizer yang telah dicoba, beserta akurasinya. Di antara kombinasi-kombinasi tersebut, terdapat nilai akurasi terbaik yang diperoleh ketika menggunakan optimizer 'Adadelta', yaitu sekitar 59.00%. Selain itu, juga ditampilkan kombinasi optimizer lainnya beserta akurasinya, seperti 'RMSprop', 'Adamax', 'Adagrad', 'SGD', dan masing-masing memiliki akurasi yang berbeda-beda.

Dalam eksperimen ini, optimizer 'Adadelta' menghasilkan kinerja model terbaik dibandingkan dengan kombinasi optimizer lainnya yang telah dicoba.

7.2.3. Bayesian Optimization

Bayesian Optimization adalah teknik lain untuk melakukan hyperparameter tuning. Dibandingkan dengan grid search dan random search, Bayesian Optimization lebih efisien karena menggunakan konsep probabilitas untuk mencari hyperparameter terbaik.

Proses Bayesian Optimization melibatkan langkah-langkah berikut:

1. Mendefinisikan fungsi objektif: Tentukan fungsi objektif yang ingin dioptimalkan. Dalam konteks ini, biasanya merupakan fungsi kinerja model (misalnya, akurasi, presisi, atau recall).
2. Memilih ruang pencarian (search space): Tentukan rentang atau batasan hyperparameter yang ingin dioptimalkan. Misalnya, untuk learning rate, kamu dapat menentukan bahwa nilainya harus antara 0.001 dan 0.1.
3. Menginisialisasi model probabilistik: Bayesian Optimization menggunakan model probabilistik untuk memodelkan fungsi objektif. Pada awalnya, model tersebut diinisialisasi dengan beberapa titik sampel dari ruang pencarian.
4. Melakukan iterasi: Selama beberapa iterasi, model probabilistik digunakan untuk memperkirakan di mana hyperparameter berada yang kemungkinan besar menghasilkan nilai kinerja yang lebih baik. Kemudian, dilakukan eksplorasi dan eksploitasi untuk menentukan titik sampel berikutnya yang harus dievaluasi.
5. Evaluasi model: Pada setiap iterasi, model probabilistik memberikan prediksi tentang performa hyperparameter selanjutnya yang harus dievaluasi. Model tersebut kemudian dievaluasi dengan melatih dan menguji model dengan hyperparameter tersebut untuk mendapatkan nilai kinerja aktual.
6. Update model: Setelah mendapatkan hasil evaluasi, model probabilistik diperbarui untuk memasukkan data baru dan meningkatkan perkiraan tentang fungsi objektif.
7. Konvergensi: Proses ini terus berlanjut sampai mencapai kondisi berhenti, misalnya ketika telah mencapai jumlah iterasi yang ditentukan atau ketika nilai kinerja yang cukup baik telah ditemukan.

Berikut adalah contoh implementasi Bayesian Optimization dengan menggunakan library Hyperopt:

```
# Install the necessary libraries first
# pip install hyperopt
# pip install numpy

import numpy as np
from hyperopt import fmin, tpe, hp

# Defining the objective function to optimize
```

```
def objective_function(params):
    # Simulate some synthetic data (replace this with your actual data)
    np.random.seed(42)
    data = np.random.normal(loc=params['mean'], scale=params['std'],
size=100)

    # Compute the metric you want to optimize (e.g., mean squared error)
    metric = np.mean(data**2)  # Replace this with your actual metric

    return metric

# Define the hyperparameter search space
space = {
    'mean': hp.uniform('mean', -10, 10),  # Uniformly sample mean from
-10 to 10
    'std': hp.uniform('std', 1, 5)        # Uniformly sample std from 1
to 5
}

# Run Bayesian Optimization
best = fmin(fn=objective_function, space=space, algo=tpe.suggest,
max_evals=100)

# Print the best hyperparameters found
print("Best hyperparameters:")
print(best)
```

Output:

```
100%|██████████| 100/100 [00:01<00:00, 85.23trial/s, best loss: 1.0455352073251276]
Best hyperparameters:
{'mean': -0.004750697384663299, 'std': 1.1235725077789955}
```

Best hyperparameters: {'mean': -0.004750697384663299, 'std': 1.1235725077789955}: Hasil ini menampilkan hyperparameter terbaik yang ditemukan oleh proses Bayesian Optimization. Dalam contoh ini, nilai hyperparameter 'mean' yang optimal adalah sekitar -0.00475, sementara nilai 'std' yang optimal adalah sekitar 1.12357. Hyperparameter ini dipilih berdasarkan hasil evaluasi dari fungsi objective (mean squared error) selama proses Bayesian Optimization, yang mengarahkan ke nilai terbaik untuk hyperparameter tersebut.

7.3. Menggunakan Pretrained Models

Model pretrained merupakan bagian penting dalam pengembangan solusi berbasis deep learning, termasuk NLP. Model ini telah dilatih terlebih dahulu pada dataset yang sangat besar, seperti dataset Wikipedia atau Common Crawl, dan biasanya memiliki performa yang

baik dalam mengekstraksi fitur-fitur penting dari data. Dalam konteks NLP, model-model pretrained populer termasuk BERT, GPT, dan Transformer.

Pada subab ini, kita akan membahas bagaimana menggunakan model pretrained untuk tugas NLP dalam Bahasa Indonesia, dengan memberikan contoh konkret menggunakan model BERT.

7.3.1. Menggunakan Model Pretrained

Model pretrained seperti BERT memiliki kemampuan untuk memahami struktur dan makna dari teks bahasa alam, sehingga mereka dapat digunakan untuk berbagai tugas NLP seperti klasifikasi teks, analisis sentimen, atau named entity recognition.

Untuk menggunakan model pretrained, kamu pertama-tama perlu memuat model dan tokenizer. Tokenizer digunakan untuk mengubah teks input menjadi format yang dapat dimengerti oleh model.

Berikut adalah contoh menggunakan model BERT yang telah dilatih pada data Bahasa Indonesia:

```
import torch
from transformers import BertModel, BertTokenizer

# Load the BERT model and tokenizer
model = BertModel.from_pretrained('indobenchmark/indobert-base-p1')
tokenizer =
BertTokenizer.from_pretrained('indobenchmark/indobert-base-p1')

# Function to get BERT embeddings for a given text
def get_bert_embeddings(text):
    inputs = tokenizer(text, return_tensors='pt')
    with torch.no_grad():
        outputs = model(**inputs)
    return outputs.last_hidden_state

# Example usage
text = "Ini adalah contoh teks dalam Bahasa Indonesia."
embeddings = get_bert_embeddings(text)

# The 'embeddings' variable now contains the BERT embeddings for the
given text
# You can use these embeddings for further downstream tasks or analyses
```

Output:

```
tensor([[[ 0.7713,  1.6570,  0.2810,  ...,  0.2020, -0.0965,  0.0170],
         [-0.2120, -1.4905,  0.4554,  ..., -0.3323, -0.2388,  0.4690],
         [-0.7670, -0.7745,  0.0451,  ...,  1.6517, -0.3380,  0.0786],
         ...,
         [-0.9579,  0.3391,  0.5636,  ...,  0.3314, -0.6540,  0.0902],
         [-0.3022,  0.0542,  0.5436,  ...,  1.5595, -0.2862,  1.2871],
         [ 0.0747,  0.0045,  0.5381,  ...,  1.0957, -0.5635,  0.9610]]])
```

Output dari script di atas adalah representasi BERT (embeddings) dari teks yang dimasukkan. Representasi ini merupakan representasi vektor yang menggambarkan makna dari teks dalam ruang semantik yang lebih tinggi. Representasi ini sangat berguna untuk berbagai tugas pemrosesan bahasa alami, seperti klasifikasi, analisis sentimen, pertanyaan jawaban, dan banyak lagi.

Sebagai contoh sederhana, kita akan menggunakan representasi BERT untuk menghitung similaritas kosinus antara dua teks. Kami akan menggunakan modul cosine_similarity dari library sklearn.metrics.pairwise untuk ini. Pastikan kamu telah menginstal library scikit-learn dengan menjalankan perintah pip install scikit-learn sebelum menjalankan script berikut:

```
from sklearn.metrics.pairwise import cosine_similarity

# Example usage
text1 = "Ini adalah contoh teks dalam Bahasa Indonesia."
text2 = "Teks ini berbicara tentang alam dan lingkungan."

embeddings1 = get_bert_embeddings(text1)
embeddings2 = get_bert_embeddings(text2)

# Calculate cosine similarity between the embeddings
similarity = cosine_similarity(embeddings1.squeeze().numpy(),
embeddings2.squeeze().numpy())

print(f"Cosine similarity between the two texts: {similarity[0][0]}")
```

Output:

```
Cosine similarity between the two texts: 0.862989068031311
```

7.3.2. Transfer Learning

Salah satu keuntungan utama dari model pretrained adalah kemampuan mereka untuk digunakan dalam skenario transfer learning. Dalam transfer learning, pengetahuan yang diperoleh saat melatih model pada satu tugas digunakan untuk membantu melatih model pada tugas yang lain.

Misalnya, kita dapat menggunakan model BERT yang telah dilatih pada data Bahasa Indonesia untuk tugas klasifikasi teks. Pertama-tama, kita memuat model dan tokenizer seperti sebelumnya. Kemudian, kita menambahkan layer tambahan pada bagian atas model untuk tugas klasifikasi, dan melatih layer ini pada data kita.

Berikut adalah contoh melakukan transfer learning dengan BERT untuk tugas klasifikasi teks:

```
from transformers import BertForSequenceClassification, BertTokenizer
from torch.optim import Adam
from torch.nn import CrossEntropyLoss
import torch
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score

# Load the BERT model and tokenizer
model =
BertForSequenceClassification.from_pretrained('indobenchmark/indobert-ba
se-p1', num_labels=2)
tokenizer =
BertTokenizer.from_pretrained('indobenchmark/indobert-base-p1')

# Prepare the data
texts = [
    "Produk ini sangat bagus! Saya sangat menyukainya.",
    "Tidak puas dengan kualitas produk ini. Buruk sekali.",
    "Pengiriman sangat cepat, pelayanan yang baik.",
    "Saya kecewa dengan produk ini. Tidak sesuai ekspektasi.",
    "Paket rusak saat tiba, harap perbaiki packaging.",
    "Pelayanan pelanggan sangat ramah dan membantu.",
    "Barang tidak sesuai gambar, kurang puas.",
    "Harga terlalu mahal untuk kualitas yang diberikan.",
    "Produk datang tepat waktu, terima kasih!",
    "Pengiriman lambat, butuh perbaikan.",
    "Tidak ada masalah dengan pesanan saya.",
    "Produk berkualitas tinggi, sangat puas.",
    "Kurir kurang profesional, perlu pelatihan lebih lanjut.",
    "Produk ini sudah rusak saat tiba, sangat kecewa.",
    "Packing rapi dan aman, bagus sekali.",
    "Tidak bisa memberikan bintang penuh karena ada cacat kecil pada
produk.",
    "Pelayanan yang buruk, staf tidak ramah.",
    "Produk ini tidak sesuai dengan deskripsi.",
    "Pengemasan buruk, produk bisa rusak saat pengiriman.",
    "Pengiriman tepat waktu, layanan yang baik.",
```

```
    "Kualitas produk di bawah ekspektasi."
]
labels = [1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0]

# Split the data into training and testing sets
train_texts, test_texts, train_labels, test_labels =
train_test_split(texts, labels, test_size=0.2, random_state=42)

# Convert the data into the format expected by the model
train_inputs = tokenizer(train_texts, return_tensors='pt', padding=True,
truncation=True)
train_labels = torch.tensor(train_labels)
train_inputs['labels'] = train_labels

test_inputs = tokenizer(test_texts, return_tensors='pt', padding=True,
truncation=True)
test_labels = torch.tensor(test_labels)
test_inputs['labels'] = test_labels

# Train the model
optimizer = Adam(model.parameters(), lr=1e-5)
loss_fn = CrossEntropyLoss()

for epoch in range(10):
    optimizer.zero_grad()
    outputs = model(**train_inputs)
    loss = loss_fn(outputs.logits, train_labels)
    loss.backward()
    optimizer.step()

print("Training complete.")

# Evaluation on test data
with torch.no_grad():
    test_outputs = model(**test_inputs)

predicted_labels = torch.argmax(test_outputs.logits, dim=1)

# Calculate evaluation metrics
accuracy = accuracy_score(test_labels.numpy(), predicted_labels.numpy())
precision = precision_score(test_labels.numpy(),
predicted_labels.numpy())
recall = recall_score(test_labels.numpy(), predicted_labels.numpy())
f1 = f1_score(test_labels.numpy(), predicted_labels.numpy())

print(f"Accuracy: {accuracy:.4f}")
```

```
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
```

Output:

```
Training complete.
Accuracy: 0.8000
Precision: 1.0000
Recall: 0.5000
F1-Score: 0.6667
```

Dalam contoh ini, kita telah menggunakan 21 contoh teks dengan label yang sesuai (positif atau negatif). Data dibagi menjadi data training dan testing dengan perbandingan 80:20. Kemudian, data tersebut diubah ke dalam format yang dapat digunakan oleh model BERT untuk latihan dan evaluasi.

7.4. Regularisasi dan Normalisasi

Regularisasi dan normalisasi merupakan dua teknik penting yang digunakan dalam deep learning, termasuk dalam NLP, untuk mencegah overfitting dan mempercepat proses pelatihan.

7.4.1. Regularisasi

Regularisasi adalah teknik untuk mencegah overfitting dengan menambahkan suatu hukuman terhadap kompleksitas model dalam fungsi loss. Dengan regularisasi, model diharapkan dapat mempelajari pola yang ada dalam data tanpa menghafal data tersebut.

Ada beberapa jenis regularisasi yang biasa digunakan dalam deep learning, termasuk L1 regularization, L2 regularization, dan dropout.

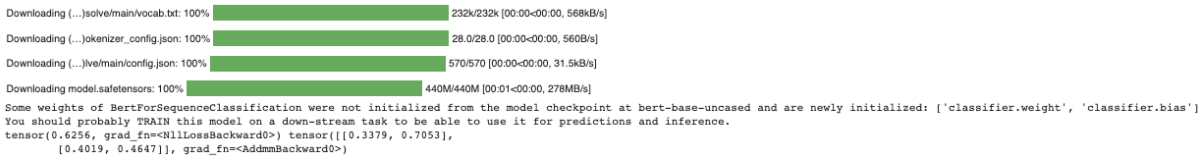
L1 dan L2 Regularization
L1 dan L2 adalah dua metode regularisasi yang paling umum digunakan. L1 regularization bekerja dengan menambahkan nilai absolut dari bobot ke fungsi loss, sedangkan L2 regularization menambahkan kuadrat dari bobot. Ini membuat bobot menjadi lebih kecil dan model menjadi lebih sederhana, sehingga membantu mencegah overfitting.

Dropout
Dropout adalah teknik regularisasi yang bekerja dengan secara acak menonaktifkan sejumlah neuron pada layer tertentu saat proses pelatihan. Ini membuat model menjadi lebih robust terhadap noise dan mencegah overfitting.

Berikut adalah contoh penggunaan regularisasi dalam model NLP:

```
# !pip install transformers

import torch
from transformers import BertTokenizer, BertForSequenceClassification
from torch.nn import Dropout

# Inisialisasi tokenizer dan model pretrained
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model =
BertForSequenceClassification.from_pretrained('bert-base-uncased')

# Menambahkan regularisasi Dropout ke model BERT
model.dropout = Dropout(0.5)  # Anda dapat mengatur rate sesuai
kebutuhan

# Contoh data (anda dapat mengganti dengan data anda sendiri)
kalimat = ["I love machine learning", "BERT is an amazing model!"]
labels = torch.tensor([1, 0]).unsqueeze(0)  # Contoh label

# Tokenisasi dan konversi kalimat ke tensor
inputs = tokenizer(kalimat, return_tensors="pt", padding=True,
truncation=True, max_length=128)
inputs["labels"] = labels

# Melakukan forward pass
outputs = model(**inputs)

# Hasilnya
loss = outputs.loss
logits = outputs.logits

print(loss, logits)
```

Output:



Penjelasan

1. torch: Pustaka untuk deep learning yang digunakan sebagai backend dari transformers.
2. BertTokenizer: Digunakan untuk mengonversi teks menjadi token-token yang dapat dimengerti oleh model BERT.

3. BertForSequenceClassification: Versi dari model BERT yang telah dilatih untuk klasifikasi sekuensial (seperti klasifikasi teks).
4. Dropout: Layer dropout yang digunakan sebagai teknik regularisasi.
5. Inisialisasi Tokenizer dan Model: tokenizer diinisialisasi dengan versi dasar dari model BERT ('bert-base-uncased'). Model juga diinisialisasi dengan versi yang sama dari BERT.
6. Menambahkan Regularisasi Dropout: Regularisasi Dropout ditambahkan ke model dengan rate 0.5. Ini berarti selama pelatihan, 50% dari neuron akan dinonaktifkan secara acak pada setiap iterasi.
7. Persiapan Data: Dua kalimat contoh diberikan bersama dengan label mereka. Dalam contoh ini, kita asumsikan label 1 berarti positif dan 0 berarti negatif. Kalimat-kalimat tersebut kemudian di-tokenisasi menggunakan tokenizer BERT untuk mengonversi teks menjadi format yang dapat diproses oleh model.
8. Forward Pass: Kalimat yang telah di-tokenisasi diteruskan ke model dengan metode forward pass. Model kemudian mengembalikan loss (kerugian) dan logits.
9. loss: Kerugian dari prediksi model dibandingkan dengan label asli.
10. logits: Nilai keluaran dari model sebelum diterapkan fungsi aktivasi (dalam kasus ini, sigmoid atau softmax).

7.4.2. Normalisasi

Normalisasi adalah teknik untuk mengubah data input menjadi format yang lebih mudah untuk model proses. Dalam konteks NLP, normalisasi bisa berarti berbagai hal, mulai dari lowercasing teks, menghapus punctuation, hingga teknik yang lebih canggih seperti tokenization atau stemming.

Berikut adalah contoh melakukan normalisasi teks:

```
import re
import spacy

# Inisialisasi model Bahasa Inggris dari spaCy (karena tidak ada model
khusus Bahasa Indonesia)
nlp = spacy.load("en_core_web_sm")

def normalize_text(text):
    # Lowercasing
    text = text.lower()

    # Menghapus tanda baca
    text = re.sub(r'[^\w\s]', '', text)

    # Tokenization
    doc = nlp(text)
    tokens = [token.text for token in doc]
```

```
    return tokens

# Contoh
text = "Halo, saya belajar NLP dengan Bahasa Indonesia!"
normalized_text = normalize_text(text)
print(normalized_text)
```

Output:

```
['halo', 'saya', 'belajar', 'nlp', 'dengan', 'bahasa', 'indonesia']
```

Dalam skrip di atas, kita menggunakan model Bahasa Inggris dari spaCy untuk tokenization karena tidak ada model khusus Bahasa Indonesia. Meskipun tidak sempurna, model ini dapat bekerja cukup baik untuk tokenization dasar.

Bab 8. Penutup

8.1. Refleksi dan Kesimpulan

Deep Learning telah menunjukkan dampak yang sangat signifikan dalam dunia teknologi dan penelitian. Sejak awal munculnya konsep neural networks hingga model-model canggih seperti transformers, revolusi yang dibawa oleh Deep Learning telah memungkinkan perkembangan inovasi dalam berbagai bidang seperti pengenalan gambar, pemrosesan bahasa alami, dan banyak lagi. Melalui buku ini, kamu diharapkan mendapatkan wawasan mendalam tentang konsep dasar, aplikasi, serta best practices dalam implementasi deep learning.

8.2. Tantangan dan Masa Depan Deep Learning

Meskipun Deep Learning telah mencapai banyak pencapaian, masih ada berbagai tantangan yang perlu diatasi. Beberapa isu seperti interpretasi model, ketahanan terhadap adanya data noise, dan kebutuhan komputasi yang besar menjadi tantangan utama dalam penelitian saat ini. Namun, dengan kemajuan teknologi dan riset yang berkelanjutan, kita dapat optimis bahwa solusi untuk isu-isu tersebut akan ditemukan.

Masa depan Deep Learning tampak cerah. Dengan kombinasi dari penelitian teoritis dan aplikasi praktis, kita mungkin akan melihat kemajuan yang lebih cepat dalam bidang medis, otomasi, energi, dan lainnya.

8.3. Tips Belajar Deep Learning Lebih Jauh

Sumber Belajar: Ada banyak kursus online, buku, dan tutorial yang membahas Deep Learning. Platform seperti Coursera, Udacity, dan edX menawarkan kursus berkualitas dengan instruktur terkemuka di bidangnya.

Proyek Praktik: Cobalah untuk mengerjakan proyek pribadi. Hal ini tidak hanya meningkatkan pemahamanmu, tetapi juga membangun portofolio yang dapat kamu tunjukkan kepada calon pemberi kerja atau rekan-rekanmu.

Bergabung dengan Komunitas: Ada banyak forum dan grup di media sosial di mana kamu bisa bertanya, berbagi, dan mendapatkan informasi terbaru tentang Deep Learning, seperti Data Science Indonesia, Datasans, dll.

Kode, Kode, Kode: Seperti semua keterampilan teknis, praktik adalah kunci. Semakin banyak kamu mengkode, semakin baik kamu memahaminya.

Kekuatan Matematika: Mempelajari dasar-dasar matematika seperti statistik, aljabar linear, dan kalkulus akan memberimu keuntungan saat memahami konsep-konsep yang lebih mendalam.

Terima Kasih