TUGAS UAS MACHINE LEARNING

Diajukan untuk memenuhi tugas pengganti Ujian Akhir Semester (UAS)

pada mata kuliah Machine Learning

Disusun oleh :

Wening Alfina Rosunika

1103204017

PROGRAM STUDI TEKNIK KOMPUTER

FAKULTAS TEKNIK ELEKTRO

UNIVERSITAS TELKOM

2023

[Gambaran Umum tentang Deep Learning dan Pytorch]
Deep learning adalah salah satu cabang dalam bidang machine learning yang memiliki fokus pada pengembangan dan penggunaan neural network, yaitu model komputasi yang digunakan untuk mempelajari representasi yang mendalam dari data. Neural network terdiri dari banyak lapisan yang saling terhubung, dan setiap lapisan melakukan transformasi pada input yang diterima untuk menghasilkan output.

Pelatihan neural network dalam deep learning dilakukan dengan melakukan proses **forward pass** dan **backward pass**. Pada tahap **forward pass**, data input diteruskan melalui jaringan saraf untuk menghasilkan prediksi output. Pada tahap **backward pass**, gradien kesalahan dihitung dan digunakan untuk memperbarui bobot dan parameter dalam jaringan melalui algoritma optimisasi seperti **stochastic gradient descent**.

Deep learning memiliki kemampuan dalam mempelajari representasi yang kompleks dari data dengan membangun hierarki fitur yang semakin kompleks melalui kombinasi lapisan dan neuron yang dalam. Hal ini memungkinkan deep learning untuk mencapai tingkat akurasi dan kinerja yang tinggi dalam berbagai tugas.

Keberhasilan deep learning telah membawa dampak besar dalam berbagai bidang seperti visi komputer, pengenalan suara, pemrosesan bahasa alami, dan lainnya. Namun, deep learning juga memiliki tantangan seperti membutuhkan volume data yang besar, kompleksitas komputasi yang tinggi, dan masalah interpretabilitas.

Secara keseluruhan, deep learning telah membawa perubahan signifikan dalam kemampuan komputasi dan aplikasi kecerdasan buatan. Dengan terus melakukan penelitian dan inovasi, deep learning diharapkan dapat memberikan dampak yang lebih besar dan membantu memahami serta memanfaatkan potensi data dengan lebih baik.

PyTorch adalah sebuah framework yang populer untuk komputasi tensor dan perhitungan diferensiasi dalam machine learning dan deep learning. PyTorch menawarkan banyak fitur dan kegunaan yang berguna dalam pengembangan model dan riset di bidang kecerdasan buatan. PyTorch memungkinkan kita untuk dengan mudah membangun, melatih, dan menerapkan model dengan menggunakan tensor, perhitungan diferensiasi otomatis, dan dukungan GPU. Dengan fitur-fitur ini, PyTorch menjadi alat yang kuat dan fleksibel untuk eksplorasi dan implementasi berbagai konsep dan teknik dalam machine learning dan deep learning.

[Judul]
Tensor Basics in PyTorch: **A Foundation for Automatic Differentiation**

[Abstrak]
Laporan teknis ini membahas konsep dasar tensor dalam PyTorch dan menggambarkan bagaimana konsep ini membantu dalam perhitungan diferensiasi otomatis. Tensor adalah struktur data inti dalam PyTorch dan merupakan representasi utama untuk komputasi numerik. Melalui penjelasan tentang pembuatan tensor, operasi dasar, dan konversi antara tensor PyTorch dan array NumPy, laporan ini memberikan pemahaman yang komprehensif tentang penggunaan tensor dalam perhitungan diferensiasi otomatis.

[Kata Kunci]
PyTorch, tensor, perhitungan diferensiasi otomatis, operasi tensor, konversi tensor

1. Pendahuluan
PyTorch adalah framework machine learning yang populer yang menawarkan dukungan penuh untuk perhitungan tensor dan diferensiasi otomatis. Konsep tensor adalah inti dari PyTorch dan memungkinkan representasi data numerik dalam berbagai dimensi. Laporan ini memberikan pemahaman mendalam tentang tensor dalam PyTorch dan menjelaskan bagaimana tensor membantu dalam perhitungan diferensiasi otomatis.

2. Pembuatan Tensor
Pada bagian ini, kita menjelaskan cara membuat tensor dalam PyTorch menggunakan fungsi-fungsi seperti `torch.empty()`, `torch.rand()`, dan `torch.zeros()`. Kami menunjukkan contoh tensor dengan berbagai dimensi seperti scalar, vector, matrix, dan tensor dengan dimensi lebih tinggi.

3. Operasi Dasar pada Tensor
Tensor di PyTorch mendukung berbagai operasi matematika dasar seperti penjumlahan, pengurangan, perkalian, dan pembagian. Kami menjelaskan cara melakukan operasi ini pada tensor menggunakan operator matematika atau fungsi-fungsi khusus seperti `torch.add()`, `torch.sub()`, `torch.mul()`, dan `torch.div()`. Kami juga menggambarkan operasi in-place yang memodifikasi tensor dengan menggunakan trailing underscore, seperti `add_()`.

4. Slicing pada Tensor
Slicing adalah teknik untuk memperoleh subset dari tensor berdasarkan indeks tertentu. Kami menjelaskan cara melakukan slicing pada tensor menggunakan indeks kolom, indeks baris, dan indeks elemen. Kami juga menunjukkan bagaimana mendapatkan nilai aktual dari tensor dengan hanya satu elemen menggunakan `.item()`.

5. Reshaping Tensor
Reshaping adalah proses mengubah dimensi tensor. Kami menjelaskan penggunaan fungsi `torch.view()` untuk mengubah dimensi tensor menjadi bentuk yang diinginkan. Kami menggambarkan contoh penggunaan `view()` untuk mereshape tensor menjadi array 1D dan 2D.

6. Konversi Antara Tensor PyTorch dan Array NumPy
PyTorch menyediakan kemampuan untuk mengonversi tensor PyTorch menjadi array NumPy dan sebaliknya. Kami menjelaskan cara melakukan konversi ini menggunakan fungsi `.numpy()` dan `torch.from_numpy()`. Kami juga memberikan peringatan tentang pembagian memori antara tensor PyTorch dan array NumPy, terutama ketika melakukan operasi in-place.

7. Perhitungan Diferensiasi Otomatis dengan Tensor
Salah satu keunggulan utama PyTorch adalah kemampuannya untuk melakukan diferensiasi otomatis. Dalam bagian ini, kami menjelaskan konsep dasar perhitungan diferensiasi otomatis menggunakan tensor PyTorch. Kami menunjukkan cara mengaktifkan perhitungan gradien pada tensor dengan menggunakan argumen `requires_grad=True`. Hal ini mengizinkan PyTorch untuk

melacak perhitungan pada tensor dan menghasilkan gradien secara otomatis menggunakan metode *backward()*.

8. Kesimpulan
Laporan ini memberikan pemahaman komprehensif tentang penggunaan tensor dalam PyTorch dan bagaimana konsep tensor ini membantu dalam perhitungan diferensiasi otomatis. Dengan tensor sebagai dasar, PyTorch menyediakan alat yang kuat untuk mengimplementasikan dan melatih model machine learning yang kompleks dengan kemampuan diferensiasi otomatis yang efisien.

[Laporan Teknis: Autograd dalam PyTorch untuk Diferensiasi Otomatis]

1. Pendahuluan
Paket autograd dalam PyTorch menyediakan diferensiasi otomatis untuk semua operasi yang dilakukan pada tensor. Diferensiasi otomatis adalah konsep dasar dalam deep learning yang memungkinkan perhitungan gradien, sehingga memungkinkan diferensiasi yang efisien dan otomatis dalam jaringan saraf. Laporan teknis ini menjelaskan paket autograd dalam PyTorch dan bagaimana hal itu memfasilitasi diferensiasi otomatis.

2. Inisialisasi Tensor dan Pelacakan Operasi
Kode dimulai dengan inisialisasi tensor `x` dengan ukuran (3,) dan `requires_grad=True`. Flag ini mengaktifkan pelacakan semua operasi yang dilakukan pada tensor. Tensor `y` kemudian dibuat dengan menambahkan 2 ke `x`. Penting untuk dicatat bahwa `y` memiliki atribut `grad_fn` yang merujuk pada fungsi yang bertanggung jawab atas pembuatan tensor tersebut.

3. Melakukan Operasi dan Menghitung Gradien
Operasi lebih lanjut dilakukan pada `y`, seperti kuadrat dan perkalian dengan 3. Akhirnya, nilai rata-rata dari `z` dihitung. Untuk menghitung gradien, fungsi `backward()` dipanggil pada `z`, yang memicu perhitungan gradien mundur. Gradien kemudian dapat diakses melalui atribut `x.grad`, yang merupakan turunan parsial dari fungsi terhadap `x`.

4. Mengelola Tensor Non-Skalar
Pada bagian berikutnya, tensor non-skalar `x` diinisialisasi dan serangkaian operasi dilakukan padanya. Karena `y` adalah tensor non-skalar, diperlukan spesifikasi tensor gradien `v` dengan bentuk yang sesuai saat memanggil `backward(v)`. Langkah ini diperlukan untuk menghitung produk vektor-Jacobian dengan akurat.

5. Mengelola Pelacakan Gradien
Ada skenario di mana beberapa operasi tidak perlu dimasukkan dalam perhitungan gradien, seperti pembaruan bobot selama pelatihan. PyTorch menyediakan beberapa pendekatan untuk menghentikan pelacakan riwayat tensor. Ini termasuk menggunakan `x.requires_grad_(False)` untuk mengubah flag dengan cara inplace, `x.detach()` untuk membuat tensor baru tanpa perhitungan gradien, atau melapisi operasi dengan `with torch.no_grad():` untuk sementara menonaktifkan pelacakan gradien.

6. Mengosongkan Gradien dan Optimisasi
Selama optimisasi, penting untuk mengosongkan gradien sebelum melanjutkan ke langkah optimisasi berikutnya. Kode menunjukkan penggunaan `.zero_()` untuk menghapus gradien dari tensor bobot. Langkah ini memastikan bahwa gradien tidak akan terakumulasi pada iterasi selanjutnya, sehingga tidak mempengaruhi bobot dan output akhir.

7. Kesimpulan
Paket autograd dalam PyTorch memainkan peran penting dalam diferensiasi otomatis, memungkinkan perhitungan gradien yang efisien

[Laporan Teknis: Backpropagation dalam PyTorch untuk Diferensiasi Otomatis]

1. Pendahuluan
Backpropagation adalah algoritma kunci dalam deep learning yang digunakan untuk menghitung gradien bobot dalam jaringan saraf. Dalam laporan teknis ini, kami akan menjelaskan konsep backpropagation dalam konteks PyTorch dan bagaimana itu membantu dalam perhitungan diferensiasi otomatis.

2. Implementasi Regresi Linear Sederhana
Kode di atas mengimplementasikan regresi linear sederhana menggunakan PyTorch. Pertama, tensor `x` dan `y` diinisialisasi dengan nilai 1.0 dan 2.0. Selanjutnya, bobot `w` diinisialisasi dengan nilai 1.0 dan ditandai dengan `requires_grad=True`, yang mengaktifkan pelacakan gradien.

3. Perhitungan Forward
Selanjutnya, nilai yang diprediksi `y_predicted` dihitung dengan mengalikan `w` dengan `x`. Kemudian, kita menghitung loss dengan mengkuadratkan selisih antara `y_predicted` dan `y`.

4. Perhitungan Backward
Setelah perhitungan loss, kita dapat memulai perhitungan gradien menggunakan fungsi `backward()`. Fungsi ini menginisiasi perhitungan gradien mundur dengan mengaplikasikan aturan rantai untuk memperbarui gradien bobot `w`.

5. Update Bobot
Setelah perhitungan gradien, kita menggunakan blok `with torch.no_grad()` untuk mencegah pelacakan gradien saat memperbarui bobot `w`. Dalam blok ini, bobot diperbarui dengan mengurangi nilai gradien dikali dengan laju pembelajaran (0.01).

6. Mengosongkan Gradien
Setelah memperbarui bobot, kita menggunakan `w.grad.zero_()` untuk mengosongkan gradien bobot sebelum memulai iterasi berikutnya. Langkah ini diperlukan untuk memastikan bahwa gradien tidak terakumulasi dari iterasi sebelumnya dan tidak mempengaruhi hasil perhitungan selanjutnya.

7. Kesimpulan
Dalam PyTorch, backpropagation adalah proses penting dalam diferensiasi otomatis yang memungkinkan perhitungan gradien bobot secara efisien. Dengan menggunakan fungsi `backward()`, kita dapat menghitung gradien secara otomatis dan dengan mudah memperbarui bobot dalam jaringan saraf. Hal ini memungkinkan pelatihan yang efektif dan iteratif dari model deep learning.

[Laporan Teknis: Gradient Descent Manual dalam PyTorch]

1. Pendahuluan
Dalam laporan teknis ini, kita akan menjelaskan implementasi metode gradient descent manual menggunakan library PyTorch. Gradient descent adalah salah satu metode optimisasi yang digunakan dalam machine learning untuk mencari nilai minimum fungsi loss. Kode di atas mengilustrasikan konsep tersebut.

2. Implementasi Gradient Descent
Dalam kode di atas, kita menggunakan library PyTorch untuk membuat tensor `w` yang akan dioptimisasi. Tensor `w` diinisialisasi dengan nilai 1.0 dan diberi atribut `requires_grad=True` untuk menandakan bahwa kita ingin menghitung gradien terhadap tensor ini.

3. Perhitungan Forward dan Loss
Kita menggunakan tensor `x` dan `y` sebagai data input dan target. Fungsi `forward(x)` digunakan untuk menghitung prediksi `y_pred` dengan mengalikan `x` dengan tensor `w`. Fungsi `loss(y, y_pred)` digunakan untuk menghitung loss dengan memperhitungkan selisih antara `y_pred` dan `y`, dalam hal ini kita menggunakan squared loss.

4. Perhitungan Gradien dan Update Bobot
Dalam loop training, kita melakukan iterasi sebanyak `n_iters` kali. Pada setiap iterasi, kita melakukan langkah-langkah berikut:
- Menghitung prediksi `y_pred` dengan memanggil fungsi `forward(x)`.
- Menghitung loss dengan memanggil fungsi `loss(y, y_pred)`.
- Memanggil metode `backward()` pada tensor `loss` untuk menghitung gradien terhadap tensor `w`.
- Menggunakan metode `item()` pada tensor `w.grad` untuk mendapatkan nilai gradien.
- Mengupdate tensor `w` dengan menggunakan formula gradient descent: `w = w - learning_rate * w.grad`.
- Mengatur gradien tensor `w.grad` menjadi nol dengan metode `zero_()` untuk persiapan iterasi selanjutnya.

5. Manfaat dalam Perhitungan Diferensiasi Otomatis
Metode gradient descent manual dalam PyTorch membantu dalam perhitungan diferensiasi otomatis dengan cara berikut:
- Dengan mengatur atribut `requires_grad=True` pada tensor yang ingin dioptimisasi, PyTorch secara otomatis melacak semua operasi yang dilakukan pada tensor tersebut.
- Saat kita memanggil metode `backward()` pada tensor loss, PyTorch akan secara otomatis menghitung gradien terhadap tensor yang memiliki atribut `requires_grad=True`.
- Dengan adanya perhitungan gradien otomatis ini, kita dapat menggunakan metode gradient descent atau algoritma optimisasi lainnya untuk memperbarui bobot model dengan cepat dan efisien.

6. Kesimpulan
Dalam laporan teknis ini, kita telah menjelaskan implementasi metode gradient descent manual dalam PyTorch. Metode ini memungkinkan kita untuk melakukan optimisasi bobot model dengan menghitung gradien secara otomatis. Hal ini memudahkan dalam perhitungan diferensiasi otomatis dan membantu dalam pelatihan model machine learning. Dengan menggunakan PyTorch, kita dapat dengan mudah menerapkan metode optimisasi yang kuat dan efisien untuk memperbarui parameter model secara iteratif

[Laporan Teknis: Gradient Descent Otomatis dalam PyTorch]

1. Pendahuluan
Dalam laporan teknis ini, kita akan menjelaskan implementasi metode gradient descent otomatis menggunakan library PyTorch. Gradient descent adalah salah satu metode optimisasi yang digunakan dalam machine learning untuk mencari nilai minimum fungsi loss. Kode di atas mengilustrasikan konsep tersebut.

2. Implementasi Gradient Descent Otomatis
Dalam kode di atas, kita menggunakan library PyTorch untuk membuat tensor `X` sebagai variabel independen dan tensor `Y` sebagai variabel dependen. Tensor `w` diinisialisasi dengan nilai 0.0 dan diberi atribut `requires_grad=True` untuk menandakan bahwa kita ingin menghitung gradien terhadap tensor ini.

3. Perhitungan Forward dan Loss
Kita menggunakan fungsi `forward(x)` untuk menghitung prediksi `y_pred` dengan mengalikan tensor `w` dengan tensor `x`. Fungsi `loss(y, y_pred)` digunakan untuk menghitung loss dengan memperhitungkan selisih antara `y_pred` dan `y`, dalam hal ini kita menggunakan squared loss.

4. Perhitungan Gradien dan Update Bobot
Dalam loop training, kita melakukan iterasi sebanyak `n_iters` kali. Pada setiap iterasi, kita melakukan langkah-langkah berikut:
- Menghitung prediksi `y_pred` dengan memanggil fungsi `forward(X)`.
- Menghitung loss dengan memanggil fungsi `loss(Y, y_pred)`.
- Memanggil metode `backward()` pada tensor `l` (loss) untuk menghitung gradien terhadap tensor `w`.
- Menggunakan metode `item()` pada tensor `w.grad` untuk mendapatkan nilai gradien.
- Mengupdate tensor `w` dengan menggunakan formula gradient descent: `w = w - learning_rate * w.grad`.
- Mengatur gradien tensor `w.grad` menjadi nol dengan metode `zero_()` untuk persiapan iterasi selanjutnya.

5. Manfaat dalam Perhitungan Diferensiasi Otomatis
Metode gradient descent otomatis dalam PyTorch membantu dalam perhitungan diferensiasi otomatis dengan cara berikut:
- Dengan mengatur atribut `requires_grad=True` pada tensor yang ingin dioptimisasi, PyTorch secara otomatis melacak semua operasi yang dilakukan pada tensor tersebut.
- Saat kita memanggil metode `backward()` pada tensor loss, PyTorch akan secara otomatis menghitung gradien terhadap tensor yang memiliki atribut `requires_grad=True`.
- Dengan adanya perhitungan gradien otomatis ini, kita dapat menggunakan metode gradient descent atau algoritma optimisasi lainnya untuk memperbarui bobot model dengan cepat dan efisien.

6. Kesimpulan
Dalam laporan teknis ini, kita telah menjelaskan implementasi metode gradient descent otomatis dalam PyTorch. Metode ini memungkinkan kita untuk melakukan optimisasi bobot model dengan menghitung gradien secara otomatis. Hal ini memudahkan dalam perhitungan diferensiasi otomatis dan membantu dalam pelatihan model machine learning. Dengan menggunakan PyTorch, kita dapat dengan mudah menerapkan metode optimisasi yang kuat dan efisien untuk memperbarui parameter model secara iteratif.

[Laporan Teknis: Loss dan Optimizer dalam PyTorch]

1. Pendahuluan
Dalam laporan teknis ini, kita akan menjelaskan konsep dan implementasi dari Loss dan Optimizer dalam library PyTorch. Kode di atas mengilustrasikan penggunaan Loss dan Optimizer dalam sebuah model sederhana.

2. Implementasi Loss dan Optimizer
Dalam kode di atas, kita menggunakan library PyTorch untuk membuat tensor `X` sebagai variabel independen dan tensor `Y` sebagai variabel dependen. Tensor `w` diinisialisasi dengan nilai 0.0 dan diberi atribut `requires_grad=True` untuk menandakan bahwa kita ingin menghitung gradien terhadap tensor ini.

3. Perhitungan Forward
Kita menggunakan fungsi `forward(x)` untuk menghitung prediksi `y_predicted` dengan mengalikan tensor `w` dengan tensor `x`. Pada awalnya, kita mencetak prediksi sebelum pelatihan menggunakan fungsi `forward(5)`.

4. Penentuan Loss Function
Dalam kode di atas, kita menggunakan `nn.MSELoss()` (Mean Squared Error) sebagai fungsi loss. Fungsi ini mengukur selisih antara prediksi (`y_predicted`) dan target (`Y`), kemudian menghitung rata-rata kuadrat dari selisih tersebut.

5. Optimizer dan Learning Rate
Kita menggunakan optimizer `torch.optim.SGD` (Stochastic Gradient Descent) dengan learning rate (`lr`) sebesar `learning_rate`. Optimizer ini akan mengoptimalkan nilai tensor `w` dengan memperbarui bobot model berdasarkan gradien yang dihitung.

6. Training Model
Dalam loop training, kita melakukan iterasi sebanyak `n_iters` kali. Pada setiap iterasi, langkah-langkah yang dilakukan adalah:
- Menghitung prediksi `y_predicted` dengan memanggil fungsi `forward(X)`.
- Menghitung loss dengan memanggil fungsi `loss(Y, y_predicted)`.
- Memanggil metode `backward()` pada tensor `l` (loss) untuk menghitung gradien terhadap tensor `w`.
- Memanggil metode `step()` pada optimizer untuk memperbarui bobot model berdasarkan gradien.
- Mengatur gradien tensor `w.grad` menjadi nol dengan metode `zero_grad()` pada optimizer untuk persiapan iterasi selanjutnya.

7. Manfaat dalam Perhitungan Diferensiasi Otomatis
Penggunaan Loss dan Optimizer dalam PyTorch memberikan manfaat dalam perhitungan diferensiasi otomatis sebagai berikut:
- Dengan menggunakan fungsi loss yang telah disediakan oleh PyTorch, seperti Mean Squared Error (MSE), kita dapat dengan mudah mengukur kesalahan antara prediksi dan target.
- Dalam fase backward, PyTorch secara otomatis menghitung gradien loss terhadap parameter yang memiliki atribut `requires_grad=True`, seperti tensor `w`.
- Dengan menggunakan optimizer, seperti Stochastic Gradient Descent (SGD), PyTorch secara otomatis melakukan pembaruan parameter model dengan mengoptimalkan nilai tensor `w` berdasarkan gradien yang dihitung.
- Seluruh proses perhitungan gradien dan pembaruan parameter dilakukan secara otomatis oleh PyTorch, sehingga memudahkan pengguna dalam mengimplementasikan algoritma optimasi yang efisien.

8. Kesimpulan

Dalam laporan ini, kita telah membahas konsep dan implementasi Loss dan Optimizer dalam PyTorch. Penggunaan Loss dan Optimizer dalam PyTorch membantu dalam perhitungan diferensiasi otomatis dengan menyediakan fungsi loss yang dapat mengukur kesalahan prediksi, serta optimizer yang memperbarui parameter model berdasarkan gradien yang dihitung. Dengan menggunakan PyTorch, pengguna dapat mengimplementasikan algoritma optimasi dengan lebih mudah dan efisien.

[Laporan Teknis: Model, Loss, dan Optimizer dalam PyTorch]

1. Pendahuluan
Dalam laporan teknis ini, kita akan menjelaskan konsep dan implementasi Model, Loss, dan Optimizer dalam library PyTorch. Kode di atas mengilustrasikan penggunaan model linear regression dengan loss function MSE (Mean Squared Error) dan optimizer SGD (Stochastic Gradient Descent) dalam PyTorch.

2. Implementasi Model Linear Regression
Dalam kode di atas, kita menggunakan class `nn.Linear` sebagai model linear regression. Model ini terdiri dari layer linear dengan input size `input_size` dan output size `output_size`. Kita juga dapat mengimplementasikan model linear regression dengan membuat class `LinearRegression` yang mewarisi class `nn.Module` dan mendefinisikan layer-layer yang dibutuhkan. Namun, pada contoh ini, kita menggunakan `nn.Linear` langsung sebagai model.

3. Persiapan Data
Kode di atas menginisialisasi tensor `X` dan `Y` sebagai data training. `X` adalah tensor yang berisi features, sedangkan `Y` adalah tensor yang berisi target. Kita juga mencetak jumlah sampel dan jumlah fitur dalam data.

4. Prediksi Sebelum Pelatihan
Kita menggunakan tensor `X_test` sebagai input untuk melihat prediksi sebelum pelatihan. Hasil prediksi ini diperoleh dengan memanggil metode `model(X_test)`. Pada awalnya, kita mencetak hasil prediksi sebelum pelatihan dengan `f(5)`.

5. Loss Function dan Optimizer
Dalam kode di atas, kita menggunakan `nn.MSELoss()` sebagai fungsi loss (Mean Squared Error). Fungsi ini digunakan untuk mengukur selisih antara prediksi dan target. Selanjutnya, kita menggunakan optimizer `torch.optim.SGD` (Stochastic Gradient Descent) dengan learning rate (`lr`) sebesar `learning_rate`. Optimizer ini akan mengoptimalkan parameter-parameter dalam model.

6. Training Model
Dalam loop training, kita melakukan iterasi sebanyak `n_iters` kali. Pada setiap iterasi, langkah-langkah yang dilakukan adalah:
- Menghitung prediksi `y_predicted` dengan memanggil metode `model(X)`.
- Menghitung loss dengan memanggil fungsi `loss(Y, y_predicted)`.
- Memanggil metode `backward()` pada tensor loss untuk menghitung gradien terhadap parameter-model.
- Memanggil metode `step()` pada optimizer untuk memperbarui parameter-model berdasarkan gradien.
- Mengatur gradien parameter-model menjadi nol dengan metode `zero_grad()` pada optimizer untuk persiapan iterasi selanjutnya.

7. Manfaat dalam Perhitungan Diferensiasi Otomatis
Penggunaan Model, Loss, dan Optimizer dalam PyTorch memberikan manfaat dalam perhitungan diferensiasi otomatis sebagai berikut:

- Dengan menggunakan model linear regression yang telah disediakan oleh PyTorch, seperti `nn.Linear`, kita dapat dengan mudah membuat dan mengkonfigurasi model dengan layer-layer yang diperlukan.
- Dalam fase backward, PyTorch secara otomatis menghitung gradien loss terhadap parameter-model, sehingga kita tidak perlu menghitung gradien secara manual.
- Dengan menggunakan optimizer, seperti SGD, PyTorch secara otom

atis mengoptimalkan parameter-model berdasarkan gradien yang telah dihitung. Ini membantu dalam proses pelatihan model dengan mengurangi kerumitan dalam mengimplementasikan algoritma optimasi secara manual.
- PyTorch juga menyediakan berbagai fungsi loss dan optimizer yang dapat disesuaikan dengan kebutuhan pengguna. Pengguna dapat memilih fungsi loss dan optimizer yang paling sesuai dengan tugas yang dihadapi.

8. Kesimpulan
Dalam laporan ini, kita telah membahas penggunaan Model, Loss, dan Optimizer dalam PyTorch. Penggunaan model linear regression dengan fungsi loss MSE dan optimizer SGD membantu dalam perhitungan diferensiasi otomatis dengan menyediakan fungsi loss yang mengukur kesalahan prediksi dan optimizer yang memperbarui parameter-model berdasarkan gradien. Dengan menggunakan PyTorch, pengguna dapat dengan mudah mengimplementasikan model, memilih fungsi loss, dan optimizer yang sesuai untuk tugas yang dihadapi, serta memanfaatkan fitur diferensiasi otomatis yang disediakan oleh library tersebut.

[Laporan Teknis: Regresi Linear dengan PyTorch]

1. Pendahuluan
Dalam laporan ini, kami akan menjelaskan implementasi Regresi Linear menggunakan PyTorch, sebuah framework Deep Learning yang populer. Regresi Linear adalah sebuah metode statistik yang digunakan untuk memodelkan hubungan linier antara variabel dependen (output) dan variabel independen (input). PyTorch memberikan dukungan yang kuat untuk perhitungan diferensiasi otomatis, yang memungkinkan kita untuk menghitung gradien secara otomatis, sehingga mempermudah proses optimisasi model dan penyesuaian parameter secara efisien.

2. Persiapan Data
Pertama, kami mempersiapkan data kita. Kita menggunakan dataset sintetis yang dihasilkan menggunakan fungsi `make_regression` dari library scikit-learn. Dataset ini terdiri dari 100 sampel dengan satu fitur dan noise sebesar 20. Kemudian, data tersebut diubah menjadi tensor menggunakan `torch.from_numpy()` untuk digunakan dalam model PyTorch. Variabel dependen (output) juga diubah menjadi tensor dan dimodifikasi dimensinya agar sesuai dengan bentuk yang diperlukan oleh model.

3. Model Regresi Linear
Selanjutnya, kita mendefinisikan model Regresi Linear menggunakan kelas `nn.Linear` dari PyTorch. Model ini memiliki satu layer linier dengan input_size sesuai dengan jumlah fitur dan output_size sebesar 1, karena kita hanya memiliki satu variabel dependen. Model ini akan melakukan prediksi berdasarkan input yang diberikan.

4. Optimisasi Model
Kita menggunakan metode optimisasi Stochastic Gradient Descent (SGD) untuk mengoptimalkan model. Kita mendefinisikan kriteria loss menggunakan `nn.MSELoss` yang merupakan Mean Squared Error. Kemudian, kita menggunakan optimizer `torch.optim.SGD` dengan learning_rate yang telah ditentukan sebelumnya. Dalam setiap epoch, kita melakukan langkah-langkah berikut:
- Melakukan prediksi menggunakan model untuk setiap input.
- Menghitung loss antara prediksi dan nilai sebenarnya.

- Menghitung gradien loss menggunakan metode `backward()`.
- Melakukan langkah optimisasi dengan memanggil `step()` pada optimizer.
- Mengatur gradien parameter model ke nol menggunakan `zero_grad()`.

5. Pelatihan dan Evaluasi Model
Kita melatih model dengan menjalankan loop epoch sebanyak yang ditentukan. Dalam setiap epoch, kita mencetak loss yang dihasilkan setelah sejumlah epoch tertentu. Setelah pelatihan selesai, kita menghasilkan prediksi menggunakan model yang telah dilatih dan mengubahnya menjadi array numpy menggunakan `detach().numpy()`. Kemudian, kita memplot data asli dan garis regresi yang dihasilkan oleh model untuk melihat sejauh mana model dapat memodelkan hubungan linier antara input dan output.

6. Kesimpulan
Dalam laporan ini, kami telah menjelaskan implementasi Regresi Linear menggunakan PyTorch. PyTorch memberikan kemudahan dalam menghitung gradien secara otomatis, yang sangat membantu dalam proses optimisasi model dan penyesuaian parameter. Dengan menggunakan PyTorch, kita dapat membangun model Deep Learning dengan mudah dan efisien untuk berbagai tugas, termasuk

[Laporan Teknis: Regresi Logistik dengan PyTorch untuk Klasifikasi Binary]

1. Pendahuluan
Dalam laporan ini, kami akan menjelaskan implementasi Regresi Logistik menggunakan PyTorch untuk melakukan klasifikasi binary. Regresi Logistik adalah metode statistik yang umum digunakan untuk memprediksi probabilitas keanggotaan pada kelas target. PyTorch, sebuah library Deep Learning, menyediakan perhitungan diferensiasi otomatis yang memudahkan proses pelatihan dan optimisasi model.

2. Persiapan Data
Kami menggunakan dataset Breast Cancer dari library scikit-learn sebagai contoh dalam laporan ini. Dataset ini berisi fitur-fitur yang menggambarkan sel-sel pada citra biopsi dan target yang menunjukkan apakah kanker tersebut bersifat jinak atau ganas. Kami membagi dataset menjadi data latih dan data uji menggunakan metode train_test_split untuk evaluasi yang lebih baik. Selanjutnya, kami melakukan penskalaan fitur menggunakan StandardScaler untuk memastikan distribusi yang seragam. Data kemudian diubah menjadi tensor menggunakan torch.from_numpy agar dapat digunakan dalam perhitungan PyTorch.

3. Model Regresi Logistik
Kami mendefinisikan model Regresi Logistik menggunakan kelas Model yang merupakan turunan dari kelas nn.Module di PyTorch. Model ini terdiri dari satu layer linier yang menggunakan kelas nn.Linear dengan jumlah input_features sesuai dengan jumlah fitur pada dataset. Output dari model dijalankan melalui fungsi sigmoid untuk menghasilkan probabilitas kelas positif.

4. Optimisasi Model
Kami menggunakan metode optimisasi Stochastic Gradient Descent (SGD) untuk mengoptimalkan model Regresi Logistik. Langkah-langkah yang dilakukan dalam proses optimisasi adalah sebagai berikut:
- Kami mendefinisikan kriteria loss menggunakan Binary Cross Entropy Loss (nn.BCELoss) yang merupakan metode yang umum digunakan untuk klasifikasi binary.
- Kami menggunakan optimizer torch.optim.SGD dengan learning_rate yang ditentukan sebelumnya untuk mengoptimalkan parameter model.
- Dalam setiap epoch, kami melakukan langkah-langkah berikut:
  - Melakukan prediksi menggunakan model pada data latih.

- Menghitung loss antara prediksi dan target menggunakan kriteria loss yang ditentukan sebelumnya.
- Menghitung gradien loss menggunakan metode backward() untuk menghitung gradien dari parameter model.
- Melakukan langkah optimisasi dengan memanggil metode step() pada optimizer untuk memperbarui parameter model berdasarkan gradien yang dihitung.
- Mengatur gradien parameter model ke nol menggunakan metode zero_grad() untuk menghapus gradien dari batch sebelumnya.

5. Pelatihan dan Evaluasi Model
Kami melatih model dengan menjalankan loop epoch sebanyak yang ditentukan. Dalam setiap epoch, kami mencetak nilai loss yang dihasilkan setelah beberapa epoch tertentu. Setelah proses pelatihan selesai, kami menggunakan model yang telah dilatih untuk memprediksi kelas pada data uji. Kami membulatkan hasil prediksi menjadi kelas yang sesuai dan menghitung akurasi dengan membandingkannya dengan target yang sebenarnya.

6. Kesimpulan
Dalam laporan ini, kami telah menjelaskan implementasi Regresi Logistik menggunakan PyTorch untuk klasifikasi binary. PyTorch menyediakan perhitungan diferensiasi otomatis yang memudahkan proses pelatihan dan optimisasi model. Dengan menggunakan PyTorch, kami dapat dengan mudah membangun dan mengoptimalkan model Regresi Logistik untuk berbagai tugas klasifikasi binary dengan hasil yang akurat.

[Laporan Teknis: DataLoader dalam PyTorch]

1. Pendahuluan
Dalam laporan ini, kami akan menjelaskan penggunaan kelas DataLoader pada PyTorch untuk memuat data dalam proses pelatihan model. DataLoader adalah komponen kunci dalam PyTorch yang membantu dalam pengelolaan dan pemrosesan data secara efisien.

2. Dataset
Sebelum memuat data menggunakan DataLoader, kita perlu mendefinisikan kelas Dataset yang mewakili dataset yang akan digunakan dalam pelatihan. Pada contoh kode di atas, kita menggunakan kelas WineDataset yang mengimplementasikan kelas Dataset. Dataset ini digunakan untuk memuat data anggur dari file CSV.

3. Inisialisasi DataLoader
Setelah kita memiliki kelas Dataset, langkah selanjutnya adalah menginisialisasi objek DataLoader. Pada contoh kode di atas, kita menginisialisasi DataLoader menggunakan dataset WineDataset yang telah dibuat sebelumnya. DataLoader memungkinkan kita untuk mengatur parameter seperti ukuran batch, pengacakan data, dan jumlah pekerja yang digunakan untuk memuat data.

4. Memuat Data dalam Batch
DataLoader memungkinkan kita untuk memuat data dalam bentuk batch. Dalam contoh kode di atas, kita menggunakan batch_size=4, yang berarti data akan dimuat dalam batch berukuran 4. Hal ini berguna dalam pelatihan model karena memungkinkan kita untuk mengoptimalkan penggunaan sumber daya dan mempercepat proses pelatihan.

5. Iterasi Melalui DataLoader
Setelah DataLoader diinisialisasi, kita dapat melakukan iterasi melalui DataLoader untuk memperoleh batch data. Dalam contoh kode di atas, kita menggunakan loop for untuk mengiterasi DataLoader dalam dua epok. Pada setiap iterasi, kita mendapatkan batch data yang terdiri dari input (features) dan label.

6. Manfaat dalam Perhitungan Diferensiasi Otomatis
DataLoader memberikan manfaat penting dalam perhitungan diferensiasi otomatis pada PyTorch. Dalam konteks pelatihan model, DataLoader memungkinkan kita untuk mengatur dan memuat data dengan mudah dalam batch. Hal ini penting dalam perhitungan diferensiasi otomatis karena memungkinkan kita untuk mengoptimalkan penggunaan memori dan mempercepat proses pelatihan.

7. Kesimpulan
Dalam laporan ini, kami menjelaskan penggunaan DataLoader dalam PyTorch untuk memuat data dalam proses pelatihan model. DataLoader memainkan peran penting dalam mengelola dan memuat data secara efisien, memungkinkan kita untuk mengatur ukuran batch, pengacakan data, dan jumlah pekerja yang digunakan. Dengan menggunakan DataLoader, kita dapat mempercepat proses pelatihan model dan memanfaatkan fitur perhitungan diferensiasi otomatis yang ada dalam PyTorch.

[Laporan Teknis: Transformers dalam PyTorch]

1. Pendahuluan
Dalam laporan ini, kami akan menjelaskan penggunaan transformers dalam PyTorch dan bagaimana hal itu membantu dalam perhitungan diferensiasi otomatis. Transformers dalam PyTorch adalah teknik yang memungkinkan kita untuk mengubah atau memanipulasi data dalam pipeline pemrosesan sebelum atau saat memuat data ke dalam model.

2. Dataset Wine
Dalam contoh kode yang diberikan, kita menggunakan dataset Wine yang merupakan dataset numerik dengan fitur-fitur yang menggambarkan atribut dari berbagai jenis anggur. Dataset ini digunakan untuk melatih model klasifikasi menggunakan pendekatan transformers dalam PyTorch.

3. Kelas WineDataset
Kelas WineDataset merupakan subkelas dari torch.utils.data.Dataset yang digunakan untuk mengimplementasikan dataset Wine. Pada konstruktor kelas ini, kita memuat data dari file CSV menggunakan numpy dan menginisialisasi atribut-atribut seperti x_data dan y_data yang mewakili fitur dan label dari dataset. Terdapat juga atribut transform yang akan digunakan untuk transformasi data.

4. Transformasi Data
Dalam contoh ini, kita menggunakan dua transformasi data: ToTensor dan MulTransform. Transformasi ToTensor digunakan untuk mengonversi data dari numpy array menjadi tensor PyTorch. Sedangkan transformasi MulTransform digunakan untuk mengalikan faktor tertentu dengan input fitur. Kedua transformasi ini diimplementasikan sebagai kelas dengan metode __call__ yang memungkinkan mereka digunakan sebagai fungsi panggilan.

5. Penerapan Transformasi
Pada contoh kode yang diberikan, kita mengilustrasikan penerapan transformasi data pada dataset Wine. Pertama, kita membuat instance dataset tanpa transformasi dan mengakses data pertama dari dataset tersebut. Kemudian, kita membuat instance dataset dengan transformasi ToTensor dan mengakses data pertama lagi. Terakhir, kita membuat instance dataset dengan komposisi transformasi ToTensor dan MulTransform, dan mengakses data pertama kembali. Hasilnya, kita dapat melihat perbedaan dalam tipe data dan nilai fitur dan label setelah transformasi.

6. Keuntungan Dalam Perhitungan Diferensiasi Otomatis
Penggunaan transformers dalam PyTorch memberikan fleksibilitas dalam memanipulasi data sebelum atau saat memuat data ke dalam model. Dengan adanya transformasi data, kita dapat melakukan operasi pengolahan data seperti konversi tipe data, normalisasi, augmentasi, dan lainnya sebelum memasukkan data ke dalam model. Hal ini memungkinkan kita untuk mengoptimalkan data

dan meningkatkan kinerja model. Selain itu, menggunakan transformers juga mempermudah implementasi alur pemrosesan data yang kompleks dengan menggabungkan beberapa transformasi secara bersamaan.

7. Kesimpulan
Dalam laporan ini, kami menjelaskan penggunaan transformers dalam PyTorch dan bagaimana hal itu membantu dalam perhitungan diferensiasi otomatis. Transformers memungkinkan kita untuk melakukan transformasi data sebelum atau saat memuat data ke dalam model. Transformasi data ini membantu dalam memanipulasi dan memproses data dengan lebih fleksibel dan efisien, yang dapat meningkatkan kinerja model secara keseluruhan. Dengan menggunakan transformers, kita dapat memanfaatkan kekuatan PyTorch dalam perhitungan diferensiasi otomatis dengan lebih baik.

[Laporan Teknis: Softmax & Cross Entropy dalam PyTorch]

Dalam laporan ini, kami akan menjelaskan penggunaan softmax dan cross entropy dalam PyTorch serta bagaimana hal itu membantu dalam diferensiasi otomatis. Softmax dan cross entropy adalah fungsi-fungsi yang umum digunakan dalam klasifikasi dalam deep learning.

Softmax menghasilkan distribusi probabilitas dari keluaran model. Fungsi ini dapat diterapkan menggunakan numpy dan PyTorch.

Cross entropy adalah metrik untuk mengukur perbedaan antara distribusi probabilitas aktual dengan distribusi probabilitas prediksi. Fungsi ini menghitung loss antara kedua distribusi probabilitas.

Keuntungan menggunakan softmax dan cross entropy dalam PyTorch adalah kemampuan diferensiasi otomatis. PyTorch menyediakan fungsi-fungsi ini dengan perhitungan gradien otomatis, yang mendukung optimasi model melalui backpropagation.

Dengan menggunakan softmax dan cross entropy, perhitungan gradien tidak perlu diimplementasikan secara manual, sehingga memudahkan optimasi model.

Kesimpulan: Softmax dan cross entropy dalam PyTorch membantu perhitungan diferensiasi otomatis, memungkinkan pengembangan model yang lebih kompleks tanpa harus khawatir dengan perhitungan gradien secara manual.

[Fungsi Aktivasi dalam PyTorch untuk Diferensiasi Otomatis]

Fungsi aktivasi penting dalam jaringan saraf untuk menambahkan non-linearitas dan mendukung diferensiasi otomatis. PyTorch menyediakan berbagai fungsi aktivasi yang dapat diintegrasikan dengan mudah ke dalam arsitektur jaringan. Dalam kode yang diberikan, kita menjelajahi fungsi-fungsi aktivasi PyTorch yang umum:

1. Softmax: Mengubah vektor bilangan real menjadi probabilitas kelas.
2. Sigmoid: Memetakan nilai-nilai input ke rentang 0 hingga 1.
3. Tanh: Memetakan nilai-nilai input ke rentang -1 hingga 1.
4. ReLU: Mengubah nilai negatif menjadi nol dan mempertahankan nilai positif.
5. Leaky ReLU: Memperbolehkan nilai negatif kecil pada ReLU.

Fungsi-fungsi ini membantu model belajar pola-pola kompleks dan mendukung diferensiasi otomatis. Dalam contoh model jaringan saraf yang diberikan, kita lihat bagaimana fungsi-fungsi aktivasi digunakan dalam kombinasi dengan lapisan linear.

Dalam kesimpulannya, fungsi aktivasi PyTorch penting dalam membangun jaringan saraf yang efektif. Mereka memungkinkan non-linearitas dan mendukung diferensiasi otomatis, yang sangat penting dalam melatih model deep learning.

Berikut ini adalah kode yang menghasilkan plot fungsi aktivasi dalam PyTorch:
sigmoid, tanh, ReLU, Leaky ReLU, dan step.

1. Sigmoid:
Fungsi sigmoid mengubah nilai input menjadi rentang antara 0 dan 1. Digunakan dalam berbagai aplikasi dan memiliki turunan yang sederhana.

2. Tanh (Hyperbolic Tangent):
Fungsi tanh menghasilkan rentang nilai antara -1 dan 1. Digunakan dalam jaringan saraf tiruan dan memiliki turunan yang mudah dihitung.

3. ReLU (Rectified Linear Unit):
Fungsi ReLU mengubah nilai negatif menjadi nol dan mempertahankan nilai positif. Digunakan secara luas dalam jaringan saraf tiruan.

4. Leaky ReLU:
Fungsi Leaky ReLU memperkenalkan gradien non-nol untuk nilai negatif kecil. Mengatasi masalah "neuron mati" pada ReLU.

5. Step Function:
Fungsi Step menghasilkan nilai 1 untuk nilai non-negatif dan 0 untuk nilai negatif. Digunakan dalam tugas klasifikasi biner.

Melalui plot ini, karakteristik masing-masing fungsi dapat divisualisasikan. Penting untuk memilih fungsi aktivasi yang tepat dalam pengembangan model jaringan saraf tiruan. Dengan perhitungan diferensiasi otomatis di PyTorch, gradien fungsi-fungsi ini dapat dihitung secara efisien untuk pelatihan model.

Dengan memahami fungsi-fungsi aktivasi ini dan manfaatnya dalam perhitungan diferensiasi otomatis, kita dapat merancang dan melatih model jaringan saraf tiruan dengan lebih efektif menggunakan PyTorch.

Feed forward adalah konsep dasar dalam jaringan saraf tiruan yang mengalirkan data dari input ke output melalui lapisan terhubung. Dalam kode di atas, kami mengimplementasikan feed forward untuk melatih dan menguji model jaringan saraf tiruan menggunakan dataset MNIST.

Pertama, dataset MNIST dimuat dan diubah menjadi tensor menggunakan torchvision.transforms.ToTensor(). Dataset ini terdiri dari gambar-gambar angka 0 hingga 9.

Selanjutnya, model jaringan saraf tiruan dibangun menggunakan kelas NeuralNet. Model ini memiliki tiga lapisan: input, tersembunyi, dan output. Lapisan input memiliki ukuran 784, lapisan tersembunyi 500, dan lapisan output 10 sesuai dengan kelas MNIST.

Dalam metode forward() model, aliran data dilakukan. Input gambar diubah menjadi vektor menggunakan images.reshape(-1, 28*28). Kemudian, vektor input dilewatkan melalui lapisan menggunakan fungsi aktivasi ReLU. Output dari lapisan output diperoleh.

Selanjutnya, fungsi loss yang digunakan adalah nn.CrossEntropyLoss(). Optimizer yang digunakan adalah torch.optim.Adam().

Selama pelatihan, setiap batch dari data pelatihan dilewatkan melalui jaringan. Output model dibandingkan dengan label menggunakan fungsi loss untuk menghitung loss. Gradien loss terhadap parameter-model dihitung dan optimizer.step() diterapkan untuk memperbarui parameter-model.

Setelah pelatihan selesai, model diuji pada dataset pengujian. Proses feed forward dilakukan pada setiap gambar pengujian untuk memperoleh prediksi kelas. Akurasi model dihitung dengan membandingkan prediksi dengan label sebenarnya.

Dalam perhitungan diferensiasi otomatis, feed forward sangat penting. PyTorch secara otomatis melacak setiap operasi pada tensor selama feed forward, membangun graf komputasi, dan menyimpan informasi yang diperlukan untuk menghitung gradien menggunakan backpropagation. Hal ini memungkinkan kita untuk dengan mudah menghitung gradien loss terhadap parameter-model menggunakan loss.backward() dan mengoptimalkan model menggunakan algoritma optimasi.

CNN (Convolutional Neural Network) adalah salah satu jenis jaringan saraf tiruan yang sangat baik dalam memproses data berstruktur, terutama dalam konteks pengolahan citra. Dalam kode di atas, kita menggunakan CNN untuk melakukan klasifikasi gambar menggunakan dataset CIFAR-10.

Pada awalnya, kita memuat dataset CIFAR-10 dan melakukan transformasi pada gambar menggunakan fungsi transformasi dari library torchvision. Transformasi ini mencakup konversi gambar menjadi tensor dan normalisasi nilai piksel.

Selanjutnya, kita mendefinisikan arsitektur model ConvNet yang terdiri dari lapisan-lapisan konvolusi, pooling, dan lapisan-lapisan linier untuk melakukan klasifikasi. Lapisan konvolusi digunakan untuk mengekstraksi fitur-fitur penting dari gambar, yang kemudian diproses melalui fungsi aktivasi ReLU.

Dalam proses feed forward, data gambar melewati lapisan-lapisan model ConvNet. Melalui konvolusi dan pooling, fitur-fitur semakin terabstraksi. Fitur-fitur ini kemudian dikonversi menjadi vektor dan diteruskan melalui lapisan-lapisan linier dengan fungsi aktivasi ReLU. Output akhir dari model adalah prediksi kelas gambar.

Ketika melatih model, kita menggunakan fungsi loss CrossEntropyLoss dan optimisasi dengan algoritma Stochastic Gradient Descent (SGD). Dalam pelatihan, setiap batch data digunakan untuk menghitung loss, menghitung gradien loss terhadap parameter-model, dan memperbarui parameter-model menggunakan optimizer.

Setelah pelatihan selesai, model dievaluasi menggunakan dataset pengujian. Proses feed forward dilakukan pada setiap gambar pengujian untuk mendapatkan prediksi kelas. Akurasi model dihitung dengan membandingkan prediksi dengan label sebenarnya. Selain itu, akurasi klasifikasi untuk setiap kelas juga dihitung.

Dalam perhitungan diferensiasi otomatis, feed forward dalam CNN membantu dengan menghitung gradien loss terhadap parameter-model secara otomatis. PyTorch melacak setiap operasi pada tensor selama feed forward, membangun graf komputasi, dan menyimpan informasi yang diperlukan untuk menghitung gradien menggunakan algoritma backpropagation. Dengan demikian, kita dapat dengan mudah menghitung gradien dan mengoptimalkan parameter-model menggunakan algoritma optimisasi yang sesuai.

Kode di atas menunjukkan penggunaan **transfer learning** dengan model ResNet-18 yang telah dilatih sebelumnya untuk tugas klasifikasi gambar pada dataset Hymenoptera. Berikut adalah penjelasan tentang bagaimana kode tersebut menggunakan transfer learning dan diferensiasi otomatis:

1. Mengimpor Library:
   Library yang diperlukan, termasuk torch, torchvision, dan matplotlib, diimpor.

2. Pra-pemrosesan Data:
   - Transformasi data didefinisikan menggunakan fungsi `transforms.Compose`. Dua set transformasi didefinisikan untuk data pelatihan dan validasi.
   - Direktori data dan dataset gambar diinisialisasi menggunakan `datasets.ImageFolder`, yang secara otomatis memuat dan memproses gambar dari direktori yang ditentukan.
   - Data loader dibuat menggunakan `torch.utils.data.DataLoader` untuk memuat dataset dalam batch.

3. Inisialisasi Model:
   - Model ResNet-18 yang telah dilatih sebelumnya pada dataset ImageNet dimuat menggunakan `models.resnet18(pretrained=True)`.
   - Lapisan fully connected (fc) dari model ResNet-18 diganti dengan lapisan linear baru dengan dua unit output, sesuai dengan dua kelas dalam dataset Hymenoptera.
   - Model dipindahkan ke GPU (jika tersedia) menggunakan `model = model.to(device)`.

4. Fungsi Pelatihan:
   - Fungsi `train_model` didefinisikan untuk melatih model.
   - Pada setiap epoch, fungsi ini mengulang fase pelatihan dan validasi.
   - Pada setiap fase, model diatur ke mode pelatihan atau evaluasi menggunakan `model.train()` atau `model.eval()`, masing-masing.
   - Untuk setiap batch input dan label, model melakukan perhitungan maju dan mundur untuk menghitung loss dan memperbarui bobot.
   - Scheduler learning rate digunakan untuk menyesuaikan learning rate pada interval yang ditentukan.
   - Fungsi ini juga mencatat model terbaik berdasarkan akurasi validasi.

5. Melatih Model:
   - Dua contoh model ResNet-18 dibuat: `model` dan `model_conv`.
   - Lapisan fully connected dari `model` dilatih sementara lapisan konvolusi yang telah dilatih sebelumnya tetap beku.
   - Lapisan fully connected dari `model_conv` dilatih dari awal.
   - Fungsi `train_model` dipanggil untuk kedua model, menggunakan optimizer dan scheduler learning rate yang berbeda.

Dengan menggunakan transfer learning, kita memanfaatkan bobot yang telah dilatih sebelumnya dari model ResNet-18, yang telah mempelajari fitur-fitur yang kaya dari dataset ImageNet, dan mengadaptasinya ke dataset Hymenoptera. Pendekatan ini menghemat sumber daya komputasi dan memungkinkan pelatihan dengan jumlah data terbatas yang dilabeli.

Diferensiasi otomatis adalah fitur inti dari PyTorch yang memungkinkan perhitungan gradien secara otomatis. Selama perhitungan maju, PyTorch melacak operasi yang dilakukan pada tensor, membangun grafik komputasi, dan menyimpan informasi yang diperlukan untuk menghitung gradien selama perhitungan mundur. Ini mem

Tensorboard adalah alat yang digunakan untuk visualisasi dan pemantauan dalam pelatihan model. Dalam kode di atas, kita menggunakan Tensorboard dalam PyTorch untuk memvisualisasikan data gambar MNIST dan melacak metrik pelatihan seperti loss dan akurasi.

Pertama, kita menginisialisasi objek `SummaryWriter` yang akan menulis log ke direktori 'runs/mnist1'. Kemudian, kita memeriksa ketersediaan perangkat keras CUDA dan menyiapkan parameter dan hyperparameter yang diperlukan untuk pelatihan model.

Kemudian, kita memuat dataset MNIST dan membuat data loader untuk pelatihan dan pengujian. Kita juga menampilkan contoh gambar dari dataset dalam Tensorboard.

Selanjutnya, kita mendefinisikan model jaringan saraf yang terdiri dari lapisan-lapisan linear dan fungsi aktivasi ReLU. Model tersebut ditransfer ke perangkat yang sesuai dengan ketersediaan CUDA.

Selanjutnya, kita mendefinisikan fungsi kerugian (loss) dan pengoptimal yang akan digunakan dalam pelatihan model. Selain itu, kita menambahkan grafik model ke Tensorboard untuk visualisasi.

Selama pelatihan, kita melakukan loop pada setiap batch data dan melaksanakan langkah-langkah pelatihan seperti penghitungan output, perhitungan loss, dan pembaruan parameter. Selama loop tersebut, kita juga memantau loss dan akurasi secara real-time dan menulis log ke Tensorboard menggunakan `SummaryWriter`.

Setelah pelatihan selesai, kita melakukan evaluasi pada dataset pengujian dan menghitung akurasi model. Kita juga menggunakan Tensorboard untuk membuat kurva presisi-panggilan (precision-recall curve) untuk setiap kelas dalam dataset.

Tensorboard membantu dalam perhitungan diferensiasi otomatis dengan memungkinkan kita untuk memvisualisasikan metrik pelatihan dan menganalisis performa model secara interaktif. Hal ini memudahkan pemahaman tentang bagaimana model sedang belajar, memungkinkan pemantauan pelatihan secara real-time, dan memfasilitasi penyesuaian dan penyempurnaan model. Dengan menggunakan Tensorboard, kita dapat dengan mudah melacak kemajuan pelatihan, membandingkan berbagai eksperimen, dan memvisualisasikan informasi penting tentang model kita.

Dalam kode di atas, kita menggunakan fungsi `torch.save()` dan `torch.load()` dalam PyTorch untuk menyimpan dan memuat model serta status optimizer. Ini membantu dalam perhitungan diferensiasi otomatis dengan memungkinkan kita untuk menyimpan dan memulihkan model yang telah dilatih dan status optimizer yang dikonfigurasi.

1. **Penggunaan `torch.save(model, FILE)`**: Pada bagian ini, kita menggunakan `torch.save()` untuk menyimpan keseluruhan model ke dalam file "model.pth". Model yang disimpan akan mencakup struktur model dan nilai parameter yang telah dipelajari selama pelatihan.

2. **Penggunaan `torch.load(FILE)`**: Setelah model disimpan, kita dapat menggunakan `torch.load()` untuk memuat kembali model dari file yang telah disimpan. Model yang dimuat akan berisi struktur model dan nilai parameter yang sama seperti saat penyimpanan. Kita kemudian mengatur model ke mode evaluasi dengan menggunakan `loaded_model.eval()`.

3. **Penggunaan `torch.save(model.state_dict(), FILE)`**: Selain itu, kita juga dapat menyimpan hanya state dictionary (kamus status) dari model menggunakan `model.state_dict()`. Dalam contoh ini, kita menggunakan `torch.save()` untuk menyimpan state dictionary model ke dalam file "model.pth".

4. **Penggunaan `torch.load(FILE)` dan `load_state_dict()`**: Setelah state dictionary model disimpan, kita dapat memuatnya kembali menggunakan `torch.load()` dan kemudian menggunakan `load_state_dict()` untuk memuat ulang state dictionary ke model yang baru dibuat. Setelah itu, kita mengatur model ke mode evaluasi dengan `loaded_model.eval()`.

5. **Penggunaan `torch.save(checkpoint, FILE)`**: Pada bagian ini, kita menggunakan `torch.save()` untuk menyimpan checkpoint yang mencakup informasi seperti epoch terakhir, state dictionary model, dan state dictionary optimizer. Checkpoint ini disimpan ke dalam file "checkpoint.pth".

6. **Penggunaan `torch.load(FILE)` dan penggunaan state dictionary**: Setelah checkpoint disimpan, kita dapat menggunakan `torch.load()` untuk memuat checkpoint dari file "checkpoint.pth". Kemudian, kita menggunakan `load_state_dict()` untuk memuat kembali state dictionary model dan optimizer dari checkpoint. Dengan cara ini, kita dapat melanjutkan pelatihan dari titik terakhir sebelum penyimpanan.

Pemilihan penggunaan `torch.save()` atau `torch.save(model.state_dict(), FILE)` tergantung pada kebutuhan kita. Jika kita ingin menyimpan dan memuat keseluruhan model, termasuk struktur dan nilai parameter, kita dapat menggunakan `torch.save(model, FILE)`. Namun, jika kita hanya ingin menyimpan dan memuat state dictionary model, kita dapat menggunakan `torch.save(model.state_dict(), FILE)`.

Penggunaan fungsi `torch.load()` memungkinkan kita untuk memuat kembali model atau state dictionary yang telah disimpan sebelumnya. Dengan demikian, kita dapat dengan mudah memulihkan model yang telah dilatih dan konfigurasi optimizer yang sesuai, yang membantu dalam perhitungan diferensiasi otomatis dengan memungkinkan kita melanjutkan pelatihan, melakukan inferensi, atau memperoleh representasi model untuk keperluan lainnya.