TECHNICAL REPORT

DEEP LEARNING WITH PYTORCH

MUHAMMAD IQBAL DHARMAWAN
NIM 1103194125

Pengenalan Deep Learning

Pengenalan Deep Learning dimulai dengan memahami konsep dasar serta perbedaannya dengan machine learning tradisional. Deep Learning adalah subbidang dari machine learning yang menggunakan jaringan saraf tiruan dengan struktur yang lebih dalam (dalam hal ini, terdiri dari beberapa layer) untuk mempelajari representasi data secara hierarkis. Hal ini memungkinkan deep learning untuk mengekstraksi fitur yang lebih kompleks dan abstrak dari data yang diberikan.

Definisi Deep Learning

Deep Learning merupakan teknik pembelajaran mesin yang menggunakan arsitektur jaringan saraf tiruan dengan banyak layer (disebut juga deep neural network) untuk mempelajari pola dan representasi data yang kompleks. Jaringan saraf tiruan pada deep learning terdiri dari banyak neuron yang saling terhubung dan memiliki kemampuan untuk belajar secara mandiri melalui proses pelatihan.

Arsitektur jaringan saraf tiruan pada deep learning terdiri dari beberapa layer yang masing-masing terdiri dari beberapa neuron. Layer pertama disebut sebagai input layer, layer terakhir disebut sebagai output layer, dan layer di antara keduanya disebut sebagai hidden layer. Setiap neuron dalam layer terhubung dengan neuron-neuron pada layer sebelumnya dan layer sesudahnya.

Perbedaan mendasar antara deep learning dan machine learning tradisional terletak pada kemampuan deep learning dalam mempelajari representasi data secara otomatis. Pada machine learning tradisional, kita perlu secara manual mengekstraksi fitur-fitur yang relevan dari data sebelum memasukkannya ke dalam model. Sedangkan pada deep learning, model dapat belajar secara mandiri untuk mengekstraksi fitur-fitur tersebut dari data mentah. Hal ini membuat deep learning lebih efektif dalam menangani data yang kompleks dan memiliki dimensi yang tinggi.

Pytorch

PyTorch adalah sebuah framework deep learning yang populer dan open-source yang digunakan untuk mengembangkan dan melatih model deep learning. PyTorch dikembangkan oleh Facebook AI Research dan menawarkan fleksibilitas tinggi, kemudahan penggunaan, dan dukungan yang kuat untuk komputasi tensor serta algoritma diferensiasi otomatis. Berikut ini adalah penjelasan rinci mengenai PyTorch.

PyTorch juga unggul dalam sifatnya yang dinamis, memungkinkan pengguna untuk mendefinisikan grafik komputasi dengan cepat. Konstruksi grafik dinamis ini memungkinkan pengembangan model yang lebih fleksibel dan intuitif, karena memungkinkan eksekusi bersyarat, loop, dan konstruksi aliran kontrol lainnya selama pendefinisian model. Selain itu, PyTorch terintegrasi dengan baik dengan pustaka Python yang populer dan menyediakan dukungan untuk akselerasi GPU, sehingga memungkinkan komputasi yang efisien pada perangkat keras paralel.

Analisis codingan

1. Tensor Basic

kodingan tersebut mengenalkan pembuatan tensor dalam PyTorch. Tensor adalah struktur data inti dalam PyTorch dan digunakan untuk menyimpan dan memanipulasi data dalam bentuk array multidimensi. Berbagai jenis tensor dapat dibuat, seperti scalar (1D), vector (1D), matriks (2D), dan tensor dengan dimensi yang lebih tinggi.

Kemudian, dilakukan inisialisasi tensor menggunakan fungsi-fungsi seperti torch.empty(size), torch.rand(size), torch.zeros(size), dan torch.tensor(data). Setiap fungsi tersebut digunakan untuk membuat tensor dengan ukuran dan isi yang berbeda. Dalam contoh tersebut, dilakukan inisialisasi tensor dengan ukuran dan nilai acak menggunakan fungsi torch.rand(size) dan torch.zeros(size).

Selanjutnya, dijelaskan cara memanipulasi tensor seperti melakukan operasi matematika element-wise seperti penjumlahan, pengurangan, perkalian, dan pembagian. PyTorch menyediakan berbagai fungsi dan operator yang memungkinkan operasi ini, seperti torch.add(), torch.sub(), torch.mul(), dan torch.div().

Selain itu, contoh kodingan juga mencakup teknik slicing untuk mengakses subset dari tensor. Slicing digunakan untuk memperoleh bagian-bagian tertentu dari tensor, seperti baris atau kolom. Contoh ini menunjukkan penggunaan slicing dengan notasi x[:, 0], x[1, :], dan x[1, 1].

Dalam contoh kodingan, juga diperlihatkan bagaimana melakukan konversi antara tensor PyTorch dan array NumPy. PyTorch menyediakan metode .numpy() untuk mengubah tensor menjadi array NumPy dan metode .from_numpy() untuk mengubah array NumPy menjadi tensor PyTorch. Perlu diperhatikan bahwa jika tensor berada di CPU, perubahan pada tensor akan mempengaruhi array NumPy dan sebaliknya.

Terakhir, kodingan menunjukkan penggunaan GPU jika tersedia. Dengan menggunakan pernyataan torch.cuda.is_available(), kita dapat memeriksa ketersediaan GPU. Jika GPU tersedia, kita dapat memindahkan tensor ke GPU menggunakan pernyataan x.to(device) dan melakukan operasi pada GPU. Namun, perlu diperhatikan bahwa numpy tidak mendukung tensor GPU, sehingga kita perlu memindahkan tensor kembali ke CPU dengan menggunakan pernyataan to("cpu") jika ingin mengubahnya menjadi array NumPy.

Kodingan tersebut memberikan gambaran tentang cara menggunakan PyTorch untuk membuat dan memanipulasi tensor, melakukan operasi matematika, serta melakukan konversi antara tensor PyTorch dan array NumPy. Dengan PyTorch, kita dapat dengan mudah membangun dan melatih model deep learning yang kompleks.

2. Autograd

Pada awalnya, kita mendefinisikan Tensor x dengan ukuran (3) dan mengaktifkan requires_grad=True. Hal ini mengindikasikan bahwa kita ingin melacak operasi pada Tensor ini untuk menghitung gradien nanti.

Selanjutnya, kita melakukan operasi pada Tensor x dengan menambahkannya dengan 2 dan menyimpan hasilnya pada Tensor y. Pada titik ini, Tensor y memiliki atribut grad_fn yang merujuk pada fungsi yang telah membuat Tensor ini.

Kemudian, kita melakukan beberapa operasi lain pada Tensor y, seperti mengalikannya dengan dirinya sendiri tiga kali dan mengambil rata-ratanya. Operasi-operasi ini juga akan dicatat oleh autograd.

Setelah melakukan operasi, kita dapat menggunakan metode .backward() untuk menghitung gradien secara otomatis dengan backpropagation. Setelah pemanggilan .backward(), gradien dari Tensor z akan tersimpan di atribut .grad dari Tensor-tensor yang

memiliki requires_grad=True, dalam hal ini Tensor x.

Selanjutnya, kodingan menunjukkan penggunaan autograd pada Tensor non-scalar. Kita mengalikan Tensor x dengan 2 beberapa kali dan kemudian melakukan backward pass dengan tensor v sebagai argumen gradient. Hal ini diperlukan karena ukuran Tensor y lebih dari satu elemen, dan kita perlu memberikan gradient yang sesuai.

Kodingan juga mengilustrasikan beberapa cara untuk menghentikan tensor melacak histori dan gradiennya. Metode .requires_grad_(False) digunakan untuk mengubah status requires_grad sebuah Tensor secara in-place. Metode .detach() menghasilkan Tensor baru dengan konten yang sama tetapi tidak melacak gradiennya. Kita juga dapat menggunakan with torch.no_grad(): untuk melaksanakan blok kode tanpa melacak histori atau menghitung gradien.

Terakhir, kodingan menunjukkan bagaimana mengoptimasi model dengan menggunakan gradien yang dihitung oleh autograd. Dalam contoh tersebut, kita menggunakan tensor weights dan melakukan beberapa langkah optimisasi dengan memperbarui nilai bobot dan mengatur gradiennya menjadi nol menggunakan metode .zero_().

3. Backpropagation

Kodingan ini menunjukkan contoh sederhana dari proses optimisasi dengan menggunakan PyTorch. Tujuan dari optimisasi ini adalah untuk mengubah nilai parameter w agar model dapat memprediksi nilai y yang lebih akurat.

Pada awalnya, kita mendefinisikan tensor x dan y sebagai input dan output yang diinginkan. Selanjutnya, kita mendefinisikan tensor w sebagai parameter yang ingin kita optimasi dengan mengatur requires_grad=True. Hal ini memungkinkan PyTorch untuk melacak gradien terhadap tensor w.

Selanjutnya, kita melakukan forward pass untuk menghitung prediksi y_predicted dengan mengalikan w dengan x. Setelah itu, kita menghitung nilai loss dengan mengkuadratkan selisih antara y_predicted dan y.

Setelah mendapatkan nilai loss, kita melakukan backward pass dengan memanggil metode .backward(). Hal ini akan menghitung gradien loss terhadap tensor-tensor yang memiliki requires_grad=True, dalam hal ini hanya w. Setelah backward pass, gradien akan tersimpan di atribut .grad dari tensor w.

Kemudian, kita melakukan update pada nilai w menggunakan metode .grad dengan mengurangi 0.01 * w.grad. Perlu diperhatikan bahwa operasi ini tidak harus menjadi bagian dari graf komputasi, sehingga kita menggunakan torch.no_grad() untuk memastikan bahwa operasi ini tidak akan tercatat dalam graf komputasi.

Terakhir, kita perlu mengatur nilai .grad dari tensor w menjadi nol menggunakan metode .zero_(). Hal ini dilakukan untuk mempersiapkan tensor w untuk backward pass selanjutnya.

Proses optimisasi ini dapat diulang dengan melakukan forward pass, backward pass, update weights, dan pengaturan .grad menjadi nol secara berulang hingga mencapai hasil yang diinginkan.

4. gradientdescent_manually

Kodingan ini menunjukkan contoh implementasi regresi linear sederhana secara manual menggunakan NumPy. Regresi linear bertujuan untuk menemukan hubungan linear antara variabel input `X` dan variabel output `Y`.

Pada awalnya, kita mendefinisikan array `X` sebagai input dan array `Y` sebagai output yang diinginkan. Selanjutnya, kita mendefinisikan parameter `w` yang akan diupdate selama proses training. Pada tahap ini, `w` diinisialisasi dengan nilai 0.

Kemudian, kita mendefinisikan fungsi `forward(x)` yang mengimplementasikan hubungan linear antara `w` dan `x`. Fungsi ini digunakan untuk melakukan prediksi pada setiap iterasi training. Selain itu, kita juga mendefinisikan fungsi `loss(y, y_pred)` yang menghitung mean squared error (MSE) antara `y_pred` (prediksi) dan `y` (output yang diinginkan). Fungsi ini digunakan untuk mengukur sejauh mana prediksi model kita dari nilai yang sebenarnya.

Selanjutnya, kita mendefinisikan fungsi `gradient(x, y, y_pred)` yang menghitung gradien dari fungsi loss terhadap parameter `w`. Gradien ini akan digunakan untuk mengupdate nilai `w` selama proses training.

Setelah itu, kita melakukan proses training dengan menggunakan loop for sebanyak `n_iters` kali. Pada setiap iterasi, kita melakukan prediksi dengan memanggil `forward(X)`, menghitung loss dengan memanggil `loss(Y, y_pred)`, menghitung gradien dengan memanggil `gradient(X, Y, y_pred)`, dan mengupdate nilai `w` dengan mengurangi `learning_rate` dikali dengan gradien (`w -= learning_rate * dw`).

Selama proses training, kita mencetak nilai `w` dan loss setiap 2 epoch untuk melihat perubahan yang terjadi. Setelah proses training selesai, kita melakukan prediksi dengan memanggil `forward(5)` untuk melihat prediksi model kita untuk input 5.

Proses ini dilakukan secara manual dengan mengimplementasikan rumus-rumus matematis yang terkait dengan regresi linear.

5. gradientdescent_auto

Kodingan ini adalah implementasi regresi linear menggunakan PyTorch. Kode ini menggunakan fitur autograd dari PyTorch yang memungkinkan perhitungan gradien secara otomatis.

Proses training mirip dengan yang sebelumnya, tetapi perhitungan gradien dan pembaruan parameter `w` dilakukan dengan bantuan autograd.

Pada awalnya, kita mendefinisikan tensor `X` sebagai input dan tensor `Y` sebagai output yang diinginkan. Selanjutnya, kita mendefinisikan tensor `w` sebagai parameter yang akan diupdate selama proses training. Pada tahap ini, `w` diinisialisasi dengan nilai 0 dan diberi tanda `requires_grad=True` agar autograd dapat melacak operasi yang melibatkan `w` untuk menghitung gradien.

Kemudian, kita mendefinisikan fungsi `forward(x)` yang mengimplementasikan hubungan linear antara `w` dan `x`. Fungsi ini digunakan untuk melakukan prediksi pada setiap iterasi training. Selain itu, kita juga mendefinisikan fungsi `loss(y, y_pred)` yang menghitung mean squared error (MSE) antara `y_pred` (prediksi) dan `y` (output yang diinginkan). Fungsi ini digunakan untuk mengukur sejauh mana prediksi model kita dari nilai yang sebenarnya.

Selanjutnya, kita melakukan proses training dengan menggunakan loop for sebanyak `n_iters` kali. Pada setiap iterasi, kita melakukan prediksi dengan memanggil `forward(X)`, menghitung loss dengan memanggil `loss(Y, y_pred)`, dan melakukan backward pass dengan memanggil `l.backward()`. Autograd akan secara otomatis menghitung gradien loss terhadap parameter `w`.

Setelah itu, kita mengupdate nilai `w` dengan mengurangi `learning_rate` dikali dengan

gradien (`w -= learning_rate * w.grad`) menggunakan `torch.no_grad()` untuk memastikan bahwa pembaruan parameter tidak dilacak oleh autograd.

Setelah melakukan pembaruan, kita menggunakan `w.grad.zero_()` untuk mengatur ulang gradien menjadi 0 sebelum melanjutkan ke iterasi berikutnya.

Selama proses training, kita mencetak nilai `w` dan loss setiap 10 epoch untuk melihat perubahan yang terjadi. Setelah proses training selesai, kita melakukan prediksi dengan memanggil `forward(5)` untuk melihat prediksi model kita untuk input 5.

Dengan menggunakan fitur autograd dari PyTorch, kita dapat dengan mudah menghitung gradien secara otomatis tanpa perlu mengimplementasikan rumus-rumus matematis secara manual.

6. loss_and_optimizer

Kodingan ini merupakan implementasi regresi linear menggunakan PyTorch dengan pendekatan yang lebih modular. Kita menggunakan kelas `nn.Module` dari PyTorch untuk merancang model regresi linear kita.

Berikut adalah langkah-langkahnya:

1) Mendefinisikan data latihan (`X` dan `Y`) sebagai tensor.

2) Mendefinisikan kelas `LinearRegressionModel` yang merupakan turunan dari `nn.Module`. Kelas ini memiliki satu parameter `w` yang akan dioptimasi selama proses training dan memiliki metode `forward` yang mengimplementasikan fungsi regresi linear (`w * x`).

3) Membuat instance dari kelas `LinearRegressionModel` sebagai model kita.

4) Mendefinisikan fungsi loss yang digunakan, dalam kasus ini menggunakan `nn.MSELoss()` yang merupakan mean squared error loss.

5) Mendefinisikan optimizer yang digunakan, dalam kasus ini menggunakan stochastic gradient descent (SGD) dengan learning rate tertentu. Optimizer ini akan mengoptimasi parameter `w` dari model.

6) Melakukan proses training dalam loop `for` sebanyak `n_iters` kali. Pada setiap iterasi:
- Melakukan forward pass dengan memanggil `model.forward(X)` untuk melakukan prediksi.
- Menghitung loss dengan memanggil `loss(Y, y_predicted)`.
- Menghitung gradien dengan memanggil `l.backward()`.
- Melakukan pembaruan parameter dengan memanggil `optimizer.step()`.
- Mengatur ulang gradien menjadi 0 dengan memanggil `optimizer.zero_grad()`.

7) Setelah proses training selesai, kita dapat menggunakan model yang telah ditraining untuk melakukan prediksi dengan memanggil `model.forward(x)`.

Dalam pendekatan ini, kita menggunakan kelas `nn.Module` dan optimizer dari PyTorch, yang memberikan keuntungan dalam hal modularitas dan kemudahan dalam merancang model serta melaksanakan proses training.

7. model loss_and_optimizer

Pada kode yang diberikan, terdapat beberapa perubahan dan penjelasan tambahan:

1) Data latihan `X` dan `Y` sekarang memiliki bentuk matriks dengan `n_samples` baris dan `n_features` kolom. Ini memungkinkan kita untuk melatih model dengan lebih fleksibel dan dapat menangani lebih dari satu fitur.

2) Dalam contoh ini, model regresi linear yang digunakan adalah `nn.Linear` dari PyTorch. Model ini secara otomatis menginisialisasi parameter `w` dan `b` yang akan dioptimasi selama proses training. Dalam kasus ini, karena kita ingin regresi linear sederhana, kita hanya memiliki satu fitur masukan dan satu fitur keluaran.

3) Dalam loop training, kita menghitung `y_predicted` dengan memanggil `model(X)`, yang merupakan forward pass dengan model kita.

4) Fungsi loss yang digunakan tetap sama, yaitu `nn.MSELoss()` yang merupakan mean squared error loss.

5) Optimizer yang digunakan adalah stochastic gradient descent (SGD), dan kita memperbarui parameter model dengan memanggil `optimizer.step()`.

6) Setelah melakukan pembaruan parameter, kita mengatur ulang gradien menjadi 0 dengan memanggil `optimizer.zero_grad()`.

7) Di akhir proses training, kita menggunakan model yang telah dilatih untuk melakukan prediksi pada `X_test` dengan memanggil `model(X_test)`.

Pendekatan ini memanfaatkan kelas `nn.Linear` yang disediakan oleh PyTorch, yang memudahkan kita dalam merancang dan melatih model regresi linear dengan lebih efisien.

8. linear_regression

Pada kode tersebut, kita menggunakan library PyTorch untuk melatih model regresi linear pada data yang dibuat secara acak menggunakan `datasets.make_regression()` dari library scikit-learn. Berikut adalah penjelasan langkah-langkah yang terjadi dalam kode tersebut:

1) Pertama, kita mempersiapkan data latihan dengan membuat data menggunakan `make_regression()`. Data tersebut terdiri dari `n_samples` titik data dengan 1 fitur masukan dan disertai dengan labelnya. Kita kemudian mengkonversi data tersebut menjadi tensor float menggunakan `torch.from_numpy()`.

2) Selanjutnya, kita mendefinisikan model regresi linear menggunakan `nn.Linear`, yang akan memetakan fitur masukan ke fitur keluaran. Model ini memiliki 1 neuron masukan dan 1 neuron keluaran.

3) Selanjutnya, kita mendefinisikan fungsi loss yang akan digunakan untuk mengukur kesalahan prediksi model. Dalam kasus ini, kita menggunakan Mean Squared Error (MSE) loss, yang diimplementasikan dengan `nn.MSELoss()`.

4) Kita juga mendefinisikan optimizer yang akan digunakan untuk memperbarui parameter model. Dalam kasus ini, kita menggunakan stochastic gradient descent (SGD) dengan

`torch.optim.SGD`. Kita mengatur learning rate (tingkat pembelajaran) sesuai kebutuhan.

5) Di dalam loop training, kita melakukan proses forward pass dengan memasukkan data latihan ke model dan menghitung prediksi. Selanjutnya, kita menghitung loss antara prediksi dan label yang sebenarnya.

6) Setelah itu, kita melakukan proses backward pass dengan memanggil `loss.backward()` untuk menghitung gradien loss terhadap parameter-model.

7) Selanjutnya, kita melakukan pembaruan parameter dengan memanggil `optimizer.step()`, yang akan mengoptimalkan parameter model menggunakan gradien yang dihitung sebelumnya.

8) Kita mengatur ulang gradien menjadi 0 dengan memanggil `optimizer.zero_grad()` sebelum melakukan langkah berikutnya dalam loop training.

9) Setiap 10 epoch, kita mencetak loss untuk melihat kemajuan pelatihan.

10) Setelah loop training selesai, kita menghasilkan prediksi pada data latihan dengan memanggil `model(X)`. Kita mengambil nilai tensor dari prediksi menggunakan `.detach().numpy()` dan melakukan plotting data latihan (titik merah) dan prediksi model (garis biru) menggunakan `matplotlib.pyplot.plot()`.

Dengan melakukan loop training dan pembaruan parameter menggunakan SGD, model secara iteratif mempelajari hubungan antara fitur masukan dan label yang sesuai. Plot hasil prediksi memperlihatkan garis yang mendekati pola data latihan, menunjukkan bahwa model telah belajar untuk melakukan regresi linear pada data tersebut.

9. logistic_regression

Pada kode tersebut, kita menggunakan dataset breast cancer yang tersedia di library scikit-learn. Berikut adalah penjelasan langkah-langkah yang terjadi dalam kode tersebut:

1) Pertama, kita memuat dataset breast cancer menggunakan `datasets.load_breast_cancer()`. Data tersebut terdiri dari fitur-fitur dan label-labelnya.

2) Kita membagi data menjadi data latihan (80%) dan data uji (20%) menggunakan `train_test_split()` dari scikit-learn. Selanjutnya, kita melakukan penskalaan fitur menggunakan `StandardScaler()` agar memiliki mean 0 dan variansi 1.

3) Setelah data dipersiapkan, kita mengkonversi data menjadi tensor float menggunakan `torch.from_numpy()`. Kita juga mengubah dimensi tensor label menjadi (n_samples, 1).

4) Selanjutnya, kita mendefinisikan model menggunakan `nn.Linear` dan menambahkan fungsi aktivasi sigmoid di akhir. Model ini akan mengambil fitur-fitur masukan dan menghasilkan probabilitas kelas positif (menggunakan sigmoid).

5) Kita mendefinisikan fungsi loss yang akan digunakan, yaitu binary cross entropy loss (`nn.BCELoss()`) yang cocok untuk masalah klasifikasi biner.

6) Selanjutnya, kita mendefinisikan optimizer yang akan digunakan untuk memperbarui

parameter model. Dalam kasus ini, kita menggunakan stochastic gradient descent (SGD) dengan `torch.optim.SGD`. Kita mengatur learning rate (tingkat pembelajaran) sesuai kebutuhan.

7) Di dalam loop training, kita melakukan proses forward pass dengan memasukkan data latihan ke model dan menghitung prediksi probabilitas kelas positif. Selanjutnya, kita menghitung loss antara prediksi dan label yang sebenarnya.

8) Setelah itu, kita melakukan proses backward pass dengan memanggil `loss.backward()` untuk menghitung gradien loss terhadap parameter-model.

9) Selanjutnya, kita melakukan pembaruan parameter dengan memanggil `optimizer.step()`, yang akan mengoptimalkan parameter model menggunakan gradien yang dihitung sebelumnya.

10) Kita mengatur ulang gradien menjadi 0 dengan memanggil `optimizer.zero_grad()` sebelum melakukan langkah berikutnya dalam loop training.

11) Setiap 10 epoch, kita mencetak loss untuk melihat kemajuan pelatihan.

12) Setelah loop training selesai, kita menggunakan model yang telah dilatih untuk melakukan prediksi pada data uji. Kita membulatkan nilai prediksi menjadi 0 atau 1, lalu menghitung akurasi prediksi dengan membandingkan dengan label yang sebenarnya.

Dengan melakukan loop training dan pembaruan parameter menggunakan SGD, model secara iteratif mempelajari hubungan antara fitur-fitur masukan dan label kelas positif/negatif dalam dataset breast cancer. Akurasi prediksi yang dihasilkan dapat digunakan untuk mengevaluasi kinerja model pada data uji.

10. dataloader

Dalam kode di atas, kita menggunakan `DataLoader` dan membuat kelas dataset kustom (`WineDataset`) yang merupakan turunan dari kelas `torch.utils.data.Dataset`. Berikut adalah penjelasan langkah-langkah yang terjadi dalam kode tersebut:

1) Pertama, kita mendefinisikan kelas `WineDataset` yang merupakan turunan dari `torch.utils.data.Dataset`. Di dalam kelas ini, kita membaca dataset dari file CSV menggunakan NumPy, dan menyimpan fitur-fitur dan label-labelnya dalam bentuk tensor menggunakan `torch.from_numpy`. Metode `__getitem__` digunakan untuk mengakses data pada indeks tertentu, sedangkan metode `__len__` digunakan untuk mendapatkan ukuran dataset.

2) Selanjutnya, kita membuat objek dataset menggunakan `WineDataset()`. Kemudian kita dapat mengakses data pertama menggunakan `dataset[0]` dan membongkar fitur dan labelnya.

3) Kita menggunakan `DataLoader` untuk memuat seluruh dataset dengan pembagian menjadi batch-batch kecil. `DataLoader` memungkinkan kita untuk melakukan iterasi melalui dataset dalam batch-batch yang lebih kecil. Kita dapat mengatur parameter seperti ukuran batch (`batch_size`), pengacakan data (`shuffle`), dan jumlah pekerja (`num_workers`) untuk mempercepat proses pemuatan data.

4) Setelah membuat objek `DataLoader`, kita dapat mengonversinya menjadi iterator

menggunakan `iter(train_loader)` dan menggunakan `next()` untuk mengambil satu sampel acak. Ini memungkinkan kita melihat struktur dan dimensi fitur dan label dalam batch.

5) Dalam contoh training loop dummy, kita mengiterasi melalui `train_loader` dalam beberapa epoch dan beberapa iterasi. Kami mencetak dimensi dan langkah-langkah saat melatih model. Dalam contoh ini, kami menganggap total sampel dalam dataset adalah 178, dan ukuran batch adalah 4, sehingga kami memiliki 45 iterasi.

6) Selain itu, kita juga melihat contoh penggunaan `DataLoader` dengan dataset MNIST yang tersedia di torchvision.datasets. Di sini, kita menggunakan `torchvision.datasets.MNIST` untuk mengakses dataset MNIST dan kemudian menggunakan `DataLoader` untuk memuat dataset tersebut dalam batch-batch kecil.

Dengan menggunakan `DataLoader` dan dataset kustom (`WineDataset`), kita dapat mempercepat dan mempermudah proses pemuatan data, serta membagi data menjadi batch-batch yang lebih kecil untuk melatih model secara efisien.

11. Transformers

Transformasi dalam PyTorch memungkinkan kita untuk menerapkan serangkaian operasi pada data saat membuat dataset. Transformasi ini dapat diterapkan pada gambar PIL, tensor, ndarray, atau data kustom. Berikut adalah beberapa informasi penting tentang transformasi dalam PyTorch:

1) PyTorch menyediakan daftar lengkap transformasi bawaan yang dapat diterapkan pada gambar dan tensor. Anda dapat melihat daftar lengkap transformasi bawaan di halaman dokumentasi resmi PyTorch: [Built-in Transforms](https://pytorch.org/docs/stable/torchvision/transforms.html).

2) Transformasi gambar umum meliputi operasi seperti CenterCrop, Grayscale, Pad, RandomAffine, RandomCrop, RandomHorizontalFlip, RandomRotation, Resize, dan Scale.

3) Transformasi tensor umum meliputi operasi seperti LinearTransformation, Normalize, dan RandomErasing.

4) Terdapat juga transformasi konversi seperti ToPILImage yang digunakan untuk mengonversi tensor atau ndarray menjadi gambar PIL, dan ToTensor yang digunakan untuk mengonversi ndarray atau gambar PIL menjadi tensor.

5) Anda dapat menggunakan transformasi Lambda untuk menerapkan fungsi lambda kustom pada sampel.

6) Jika transformasi bawaan tidak mencukupi, Anda juga dapat menulis kelas transformasi kustom Anda sendiri dengan mengimplementasikan metode `__call__` yang menerima sampel sebagai masukan dan mengembalikan sampel yang telah diubah.

7) Anda dapat menggabungkan beberapa transformasi menggunakan `Compose`. Dalam contoh di atas, kita menggunakan `Rescale` dan `RandomCrop` sebagai contoh dalam komposisi transformasi.

Selanjutnya, dalam kode di atas, kami mengilustrasikan penggunaan transformasi pada dataset

`WineDataset` dengan membuat transformasi tensor kustom (`ToTensor`) dan transformasi kustom lainnya (`MulTransform`). Transformasi tensor kustom (`ToTensor`) mengubah ndarray menjadi tensor, sedangkan transformasi kustom lainnya (`MulTransform`) mengalikan masukan dengan faktor yang diberikan.

Hasil cetak menunjukkan perbedaan sebelum dan sesudah penerapan transformasi pada contoh dataset. Tanpa transformasi, fitur dan label tetap dalam bentuk ndarray. Namun, setelah menerapkan transformasi tensor (`ToTensor`), fitur dan label dikonversi menjadi tensor. Pada contoh terakhir, kita menggunakan komposisi transformasi dengan menerapkan transformasi tensor (`ToTensor`) dan transformasi kustom lainnya (`MulTransform`). Hasilnya, fitur pertama dikalikan dengan faktor 4.

Transformasi dalam PyTorch memungkinkan kita untuk memodifikasi data dengan mudah dan fleksibel saat membuat dataset, yang berguna dalam pra-pemrosesan data sebelum diberikan ke model.

12. Softmax and crossentropy

Dalam kode di atas, kita mengilustrasikan penggunaan softmax, cross entropy, dan fungsi loss terkait dalam NumPy dan PyTorch.

1) Fungsi softmax pada NumPy digunakan untuk menghitung probabilitas dari keluaran (output) kelas. Softmax mengaplikasikan fungsi eksponensial pada setiap elemen dan mengnormalisasi hasilnya dengan membaginya dengan jumlah eksponensial dari semua elemen tersebut. Hasilnya adalah keluaran yang terjepit antara 0 dan 1, yang mewakili probabilitas. Jumlah semua probabilitas adalah 1. Dalam kode di atas, kita menghitung softmax menggunakan NumPy pada array `x` dan mencetak hasilnya.

Dalam PyTorch, kita dapat menggunakan fungsi `torch.softmax` untuk melakukan hal yang sama pada tensor. Dalam kode di atas, kita menghitung softmax menggunakan PyTorch pada tensor `x` dan mencetak hasilnya.

2) Fungsi cross entropy pada NumPy digunakan untuk mengukur kinerja model klasifikasi yang keluarannya adalah nilai probabilitas antara 0 dan 1. Fungsi loss ini meningkat seiring perbedaan probabilitas prediksi dengan label aktual. Dalam kode di atas, kita mengimplementasikan fungsi cross entropy pada NumPy dengan memasukkan label aktual (`actual`) dan prediksi probabilitas (`predicted`).

3) Dalam PyTorch, fungsi cross entropy diimplementasikan sebagai `nn.CrossEntropyLoss()`. Fungsi ini menggabungkan langkah-langkah softmax (melalui `nn.LogSoftmax`) dan negative log likelihood loss (`nn.NLLLoss`). Dalam kode di atas, kita menggunakan `nn.CrossEntropyLoss()` untuk menghitung loss antara prediksi (`Y_pred_good` dan `Y_pred_bad`) dan label aktual (`Y`). Perhatikan bahwa input `Y_pred_good` dan `Y_pred_bad` pada `nn.CrossEntropyLoss()` adalah logit (skor sebelum softmax), bukan hasil softmax.

Selain itu, dalam kode di atas, kita juga mengilustrasikan penggunaan `nn.BCELoss()` untuk masalah klasifikasi biner (binary classification) dan `nn.CrossEntropyLoss()` untuk masalah klasifikasi multikelas (multiclass classification). Model-model jaringan saraf (neural network) sederhana juga didefinisikan menggunakan `nn.Module`.

13. Activation functions

Dalam kode di atas, kita mengilustrasikan penggunaan beberapa fungsi aktivasi dalam PyTorch.

1) Softmax: Fungsi softmax digunakan untuk menghasilkan probabilitas kelas dalam masalah klasifikasi multikelas. Fungsi softmax pada PyTorch dapat dipanggil langsung menggunakan `torch.softmax` atau dengan menggunakan objek `nn.Softmax`. Dalam kode di atas, kita menghitung softmax dari tensor `x` menggunakan kedua pendekatan tersebut.

2) Sigmoid: Fungsi sigmoid digunakan untuk menghasilkan keluaran antara 0 dan 1, yang biasanya digunakan dalam masalah klasifikasi biner. Fungsi sigmoid pada PyTorch dapat dipanggil menggunakan `torch.sigmoid` atau dengan menggunakan objek `nn.Sigmoid`. Dalam kode di atas, kita menghitung sigmoid dari tensor `x` menggunakan kedua pendekatan tersebut.

3) Tanh: Fungsi tanh (hyperbolic tangent) menghasilkan keluaran antara -1 dan 1. Fungsi tanh pada PyTorch dapat dipanggil menggunakan `torch.tanh` atau dengan menggunakan objek `nn.Tanh`. Dalam kode di atas, kita menghitung tanh dari tensor `x` menggunakan kedua pendekatan tersebut.

4) ReLU: Fungsi ReLU (Rectified Linear Unit) menghasilkan keluaran yang sama dengan inputnya jika input lebih besar dari 0, dan menghasilkan 0 jika input lebih kecil atau sama dengan 0. Fungsi ReLU pada PyTorch dapat dipanggil menggunakan `torch.relu` atau dengan menggunakan objek `nn.ReLU`. Dalam kode di atas, kita menghitung ReLU dari tensor `x` menggunakan kedua pendekatan tersebut.

5) Leaky ReLU: Fungsi Leaky ReLU adalah variasi dari ReLU yang memiliki gradien yang kecil saat input kurang dari 0. Fungsi Leaky ReLU pada PyTorch dapat dipanggil menggunakan `F.leaky_relu` atau dengan menggunakan objek `nn.LeakyReLU`. Dalam kode di atas, kita menghitung Leaky ReLU dari tensor `x` menggunakan kedua pendekatan tersebut.

Selain itu, kode di atas juga mengilustrasikan dua opsi untuk mengimplementasikan jaringan saraf dengan fungsi aktivasi. Opsi pertama adalah dengan membuat objek `nn.Module` yang memiliki modul aktivasi terpisah (misalnya, `nn.ReLU`, `nn.Sigmoid`) dan menggunakannya dalam metode `forward` jaringan saraf. Opsi kedua adalah dengan menggunakan fungsi aktivasi secara langsung dalam metode `forward` tanpa menggunakan objek `nn.Module`.

14. Plot activations

Kode di atas menghasilkan grafik fungsi aktivasi yang umum digunakan. Berikut adalah penjelasan singkat tentang setiap fungsi aktivasi yang ditampilkan dalam grafik:

1) Sigmoid: Fungsi sigmoid mengubah input menjadi output yang berada dalam rentang 0 hingga 1. Fungsi ini umum digunakan dalam masalah klasifikasi biner. Pada grafik, fungsi sigmoid ditampilkan dengan garis biru.

2) TanH: Fungsi tanh (hyperbolic tangent) menghasilkan output yang berada dalam rentang -1 hingga 1. Fungsi ini sering digunakan dalam jaringan saraf untuk memberikan keluaran yang simetris. Pada grafik, fungsi tanh ditampilkan dengan garis biru.

3) ReLU: Fungsi ReLU (Rectified Linear Unit) menghasilkan output yang sama dengan

inputnya jika input lebih besar dari 0, dan menghasilkan 0 jika input lebih kecil atau sama dengan 0. Fungsi ini sering digunakan dalam jaringan saraf untuk mengatasi masalah gradien yang hilang. Pada grafik, fungsi ReLU ditampilkan dengan garis biru.

4) Leaky ReLU: Fungsi Leaky ReLU adalah variasi dari ReLU yang memiliki gradien yang kecil saat input kurang dari 0. Hal ini membantu mengatasi masalah 'neuron mati' pada ReLU. Pada grafik, fungsi Leaky ReLU ditampilkan dengan garis biru.

5) Binary Step: Fungsi langkah biner menghasilkan output 1 jika input lebih besar atau sama dengan 0, dan output 0 jika input lebih kecil dari 0. Fungsi ini sering digunakan dalam masalah klasifikasi biner sederhana. Pada grafik, fungsi langkah biner ditampilkan dengan garis biru.

Grafik-gambar ini membantu untuk memvisualisasikan bagaimana fungsi aktivasi ini berperilaku dan bagaimana mereka mengubah input menjadi output dalam konteks fungsi matematika.

15. FeedForward

Kode di atas merupakan implementasi jaringan saraf berbasis fully connected neural network (FCNN) untuk pengenalan digit menggunakan dataset MNIST. Berikut adalah penjelasan singkat tentang kode tersebut:

1) Import library yang diperlukan: Kode tersebut mengimpor beberapa library seperti torch, torch.nn, torchvision, dan matplotlib.pyplot.

2) Konfigurasi perangkat: Kode tersebut menentukan perangkat yang akan digunakan untuk melatih model, yaitu menggunakan GPU (jika tersedia) atau CPU.

3) Menentukan hyperparameter: Kode tersebut menentukan beberapa hyperparameter seperti ukuran input, ukuran hidden layer, jumlah kelas, jumlah epoch, ukuran batch, dan learning rate.

4) Mendefinisikan dataset MNIST: Kode tersebut mendefinisikan dataset MNIST yang akan digunakan untuk melatih dan menguji model. Dataset ini disediakan oleh torchvision dan dilakukan transformasi ke format tensor menggunakan transforms.ToTensor().

5) Data loader: Kode tersebut menggunakan torch.utils.data.DataLoader untuk memuat data dalam batch-batch ke model saat melatih dan menguji.

6) Menampilkan contoh gambar: Kode tersebut menampilkan beberapa contoh gambar digit dari dataset MNIST menggunakan matplotlib.pyplot.

7) Mendefinisikan model: Kode tersebut mendefinisikan arsitektur model FCNN dengan satu layer tersembunyi. Layer pertama adalah layer linear dengan input_size dan hidden_size, diikuti oleh fungsi aktivasi ReLU, dan layer kedua adalah layer linear dengan hidden_size dan num_classes.

8) Mendefinisikan fungsi loss dan optimizer: Kode tersebut mendefinisikan fungsi loss (CrossEntropyLoss) dan optimizer (Adam) yang akan digunakan saat melatih model.

9) Melatih model: Kode tersebut melatih model dengan melakukan forward pass, menghitung

loss, melakukan backward pass, dan mengoptimasi parameter menggunakan gradient descent. Model dilatih dalam beberapa epoch dengan menggunakan data dari train_loader.

10) Menguji model: Setelah melatih model, kode tersebut menguji model pada data dari test_loader. Dalam tahap pengujian, tidak perlu menghitung gradien (untuk efisiensi memori). Akurasi model dihitung berdasarkan jumlah prediksi yang benar.

11) Menampilkan akurasi: Kode tersebut mencetak akurasi model pada 10000 gambar uji.

Implementasi ini memberikan contoh dasar tentang bagaimana menggunakan PyTorch untuk melatih model jaringan saraf untuk pengenalan digit menggunakan dataset MNIST.

16. Cnn

Kode di atas adalah implementasi jaringan saraf konvolusi (CNN) untuk klasifikasi gambar pada dataset CIFAR-10 menggunakan PyTorch. Berikut adalah penjelasan singkat tentang kode tersebut:

1) Import library yang diperlukan: Kode tersebut mengimpor beberapa library seperti torch, torch.nn, torchvision, matplotlib.pyplot, dan numpy.

2) Konfigurasi perangkat: Kode tersebut menentukan perangkat yang akan digunakan untuk melatih model, yaitu menggunakan GPU (jika tersedia) atau CPU.

3) Menentukan hyperparameter: Kode tersebut menentukan beberapa hyperparameter seperti jumlah epoch, ukuran batch, dan learning rate.

4) Transformasi data: Kode tersebut menggunakan torchvision.transforms.Compose untuk melakukan transformasi pada dataset CIFAR-10. Transformasi tersebut mengubah gambar menjadi tensor dan melakukan normalisasi.

5) Mendefinisikan dataset: Kode tersebut mendefinisikan dataset CIFAR-10 yang akan digunakan untuk melatih dan menguji model.

6) Data loader: Kode tersebut menggunakan torch.utils.data.DataLoader untuk memuat data dalam batch-batch ke model saat melatih dan menguji.

7) Menampilkan contoh gambar: Kode tersebut menampilkan beberapa contoh gambar dari dataset CIFAR-10 menggunakan matplotlib.pyplot.

8) Mendefinisikan model CNN: Kode tersebut mendefinisikan arsitektur model CNN dengan beberapa layer konvolusi, max pooling, dan fully connected layer.

9) Mendefinisikan fungsi loss dan optimizer: Kode tersebut mendefinisikan fungsi loss (CrossEntropyLoss) dan optimizer (SGD) yang akan digunakan saat melatih model.

10) Melatih model: Kode tersebut melatih model dengan melakukan forward pass, menghitung loss, melakukan backward pass, dan mengoptimasi parameter menggunakan stochastic gradient descent (SGD). Model dilatih dalam beberapa epoch dengan menggunakan data dari train_loader.

11) Menyimpan model: Setelah melatih model, kode tersebut menyimpan parameter model ke dalam file yang dapat digunakan nanti.

12) Menguji model: Kode tersebut menguji model pada data dari test_loader. Akurasi model dihitung berdasarkan jumlah prediksi yang benar.

13) Menampilkan akurasi: Kode tersebut mencetak akurasi total model dan akurasi per kelas pada dataset CIFAR-10.

Implementasi ini memberikan contoh tentang bagaimana menggunakan PyTorch untuk melatih model jaringan saraf konvolusi (CNN) untuk klasifikasi gambar pada dataset CIFAR-10.

17. transfer_learning

Kode di atas mengimplementasikan transfer learning pada model Convolutional Neural Network (CNN) menggunakan arsitektur ResNet-18 pada dataset Hymenoptera. Implementasi ini terdiri dari dua bagian utama: "Finetuning the convnet" dan "ConvNet as fixed feature extractor".

Pada bagian pertama, "Finetuning the convnet", kita menggunakan model ResNet-18 yang telah dilatih sebelumnya (pretrained) pada dataset ImageNet. Kemudian, kita mengganti layer fully connected terakhir (model.fc) dengan layer baru yang memiliki output sesuai dengan jumlah kelas pada dataset Hymenoptera (2 kelas: 'ants' dan 'bees'). Model ini kemudian dilatih kembali (finetuned) pada dataset Hymenoptera dengan menggunakan stochastic gradient descent (SGD) sebagai optimizer. Learning rate scheduler (StepLR) digunakan untuk mengurangi learning rate pada setiap langkah (step) yang ditentukan. Selama pelatihan, akurasi dan loss dicetak untuk setiap fase (train dan val), dan model dengan akurasi validasi terbaik disimpan.

Pada bagian kedua, "ConvNet as fixed feature extractor", kita menggunakan model ResNet-18 yang sama seperti sebelumnya. Namun, kali ini kita membekukan (freeze) semua parameter pada model, kecuali layer fully connected terakhir (model_conv.fc). Hal ini berarti hanya parameter pada layer terakhir yang akan diupdate selama pelatihan. Model ini juga dilatih pada dataset Hymenoptera dengan menggunakan SGD sebagai optimizer dan learning rate scheduler yang sama seperti sebelumnya.

Dengan menggunakan transfer learning, kita dapat memanfaatkan pengetahuan yang telah dipelajari oleh model pada dataset ImageNet untuk mempercepat dan meningkatkan kinerja model pada dataset Hymenoptera yang memiliki jumlah data yang lebih sedikit. Pada akhir pelatihan, model dengan akurasi validasi terbaik akan disimpan, dan hasil akhir dari kedua pendekatan transfer learning ini akan dicetak.

18. Tensorboard

Pada kode di atas, kami menggunakan TensorBoard untuk memvisualisasikan pelatihan dan evaluasi model Neural Network pada dataset MNIST. Berikut adalah langkah-langkah utama yang dilakukan:

1. Pertama, kita menginisialisasi `SummaryWriter` dengan menentukan direktori log untuk menyimpan data TensorBoard.

2. Selanjutnya, kami mendefinisikan arsitektur model Neural Network dengan menggunakan

satu layer tersembunyi (hidden layer). Model ini terdiri dari layer linear (`nn.Linear`) dengan fungsi aktivasi ReLU (`nn.ReLU`) di antara layer tersembunyi dan layer output. Model ini kemudian dipindahkan ke perangkat yang sesuai (CPU atau GPU) menggunakan `to(device)`.

3. Selanjutnya, kita mendefinisikan fungsi loss (`nn.CrossEntropyLoss`) dan optimizer (`torch.optim.Adam`) untuk melatih model.

4. Selama pelatihan, kita meloopi melalui setiap batch data dalam data loader pelatihan. Setiap batch data diteruskan ke model untuk mendapatkan output. Kemudian, kita menghitung loss dengan membandingkan output model dengan label yang benar. Selanjutnya, kita melakukan backpropagation dan update parameter model menggunakan optimizer.

5. Selama pelatihan, kita juga menghitung running loss dan running accuracy untuk setiap 100 langkah (steps). Setelah itu, kita menggunakan `writer.add_scalar` untuk mencatat running loss dan running accuracy ke TensorBoard.

6. Setelah pelatihan selesai, kita melakukan evaluasi model pada data pengujian. Kita menghitung akurasi model pada 10.000 gambar pengujian dan mencetaknya. Selain itu, kita juga mencatat kurva presisi-recall (`add_pr_curve`) untuk setiap kelas pada TensorBoard.

Terakhir, setelah semua langkah selesai, kita menutup `SummaryWriter` untuk menyimpan dan menutup log TensorBoard.

Dengan menggunakan TensorBoard, kita dapat melihat visualisasi yang berguna, seperti kurva loss, kurva akurasi, dan kurva presisi-recall, yang membantu dalam menganalisis dan memahami kinerja model secara lebih baik.

19. save_load

Dalam kode yang Anda berikan, terdapat beberapa contoh tentang cara menyimpan dan memuat model di PyTorch menggunakan `torch.save` dan `torch.load`. Berikut adalah penjelasan untuk setiap contoh:

1. Metode Pertama: Menyimpan Seluruh Model
   - Anda dapat menggunakan `torch.save` untuk menyimpan seluruh model ke file.
   - Misalnya, `torch.save(model, PATH)` akan menyimpan seluruh model ke file dengan PATH yang ditentukan.
   - Untuk memuat model yang disimpan, Anda dapat menggunakan `torch.load(PATH)`.
   - Pastikan untuk memanggil `model.eval()` setelah memuat model untuk menyetel model ke mode evaluasi.

2. Metode Kedua: Menyimpan Hanya State Dict
   - Metode ini lebih disarankan karena hanya menyimpan `state_dict` dari model, bukan seluruh model itu sendiri.
   - Anda dapat menggunakan `torch.save(model.state_dict(), PATH)` untuk menyimpan `state_dict` model ke file.
   - Untuk memuat model dari `state_dict`, Anda perlu membuat model terlebih dahulu dengan parameter yang sesuai, dan kemudian memuat `state_dict` menggunakan `model.load_state_dict(torch.load(PATH))`.
   - Setelah memuat `state_dict`, panggil `model.eval()` untuk menyetel model ke mode evaluasi.

3. Metode Ketiga: Menyimpan dan Memuat Checkpoint
   - Anda dapat menyimpan model dan status optimizer dalam satu checkpoint menggunakan dictionary.
   - Misalnya, Anda dapat membuat dictionary `checkpoint` yang berisi `'epoch'`, `'model_state'`, dan `'optim_state'`.
   - Selanjutnya, gunakan `torch.save` untuk menyimpan checkpoint tersebut ke file.
   - Untuk memuat checkpoint, Anda perlu membuat model dengan parameter yang sesuai dan optimizer yang sesuai, dan kemudian memuat `'model_state'` dan `'optim_state'` menggunakan `model.load_state_dict(torch.load(FILE)['model_state'])` dan `optimizer.load_state_dict(torch.load(FILE)['optim_state'])`.
   - Anda juga dapat memuat `'epoch'` dari checkpoint jika diperlukan.

4. Menyimpan dan Memuat pada Perangkat GPU/CPU
   - Untuk menyimpan model pada GPU dan memuatnya pada CPU, Anda perlu menetapkan perangkat saat menyimpan dan memuat menggunakan argumen `map_location`.
   - Misalnya, jika Anda menyimpan model pada GPU dengan `device = torch.device("cuda")`, Anda dapat memuatnya pada CPU dengan `model.load_state_dict(torch.load(PATH, map_location=device))`.
   - Sebaliknya, jika Anda menyimpan model pada CPU dengan `torch.save(model.state_dict(), PATH)`, Anda dapat memuatnya pada GPU dengan `model.load_state_dict(torch.load(PATH, map_location="cuda:0"))` setelah Anda menerapkan model ke perangkat GPU menggunakan `model.to(device)`.

Pastikan untuk menggunakan metode yang sesuai dengan kebutuhan Anda, terutama dalam hal menyimpan dan memuat model pada perangkat yang tepat (CPU atau GPU) untuk konsistensi dan efisiensi yang baik.