# *Technical Report: Deep Learning with Python.*

Dalam laporan ini, saya akan membahas topik "Deep Learning with PyTorch", yang merupakan kerangka kerja yang populer untuk pemrosesan data dan pembelajaran mendalam. Deep learning adalah salah satu cabang dari pembelajaran mesin yang menggunakan arsitektur jaringan syaraf tiruan untuk mempelajari pola dan fitur yang kompleks dari data.

Tujuan dibuatnya laporan ini adalah untuk memberikan pemahaman yang komprehensif tentang konsep dasar dalam deep learning dengan PyTorch. saya akan memulai dengan menjelaskan dasar-dasar tensor, yaitu struktur data fundamental dalam PyTorch. Selanjutnya, saya akan membahas topik seperti autograd, backpropagation, dan gradient descent, yang merupakan kunci dalam pelatihan model deep learning.

- **Tensor**

Pada pengembangan aplikasi berbasis deep learning, salah satu kerangka kerja yang populer adalah PyTorch. PyTorch menggunakan konsep tensor sebagai struktur data fundamental untuk pemrosesan dan pembelajaran mendalam. Tensor dalam PyTorch adalah representasi multidimensional dari data, yang dapat memiliki dimensi berbeda seperti skalar, vektor, matriks, atau tensor dengan dimensi lebih tinggi.

Dalam penggunaan tensor di PyTorch, terdapat beberapa poin penting yang perlu dipahami. Pertama, kita dapat membuat tensor menggunakan fungsi seperti `torch.empty()`, `torch.rand()`, `torch.zeros()`, dan `torch.ones()`. Fungsi-fungsi ini memungkinkan kita untuk menginisialisasi tensor dengan nilai yang diinginkan atau acak.

Selain itu, PyTorch menyediakan berbagai operasi tensor seperti penambahan, pengurangan, perkalian, dan pembagian elemen demi elemen. Kita dapat menggunakan operator matematika atau fungsi PyTorch yang sesuai untuk melakukan operasi ini. PyTorch juga mendukung operasi in-place, di mana tensor asli dapat diubah secara langsung.

Pemotongan (slicing) adalah teknik penting dalam memanipulasi tensor, yang memungkinkan kita untuk mengambil subset tertentu dari tensor. Dengan menggunakan indeks, kita dapat mengakses baris, kolom, atau elemen individu dari tensor. Selain itu, kita dapat menggunakan fungsi `view()` untuk mengubah bentuk tensor sesuai kebutuhan, baik itu merubah dimensi atau mengubah urutan elemen.

Integrasi PyTorch dengan NumPy memudahkan konversi antara tensor PyTorch dan array NumPy. Kita dapat dengan mudah mengubah tensor menjadi array NumPy menggunakan metode `.numpy()`, dan sebaliknya, mengubah array NumPy menjadi tensor menggunakan `torch.from_numpy()`. Hal ini memungkinkan kita untuk memanfaatkan kekuatan dan fleksibilitas kedua ekosistem ini dalam pengembangan aplikasi berbasis data.

Selain itu, PyTorch juga mendukung perhitungan pada GPU, yang memberikan kecepatan yang lebih tinggi dalam pemrosesan dan pelatihan model. Dengan menggunakan fungsi `torch.cuda.is_available()`, kita dapat memeriksa ketersediaan GPU pada sistem. Kemudian, kita dapat memindahkan tensor ke GPU menggunakan `.to()` untuk memanfaatkan kecepatan komputasi yang diperoleh. Jika diperlukan, kita juga dapat memindahkan tensor kembali ke CPU menggunakan `.to("cpu")`.

Penguasaan penggunaan tensor dalam PyTorch memungkinkan kita untuk melakukan pemrosesan data dan melatih model deep learning secara efisien. Integrasi dengan NumPy dan kemampuan komputasi GPU memberikan fleksibilitas dan kecepatan dalam pengembangan aplikasi deep learning.

- **Autograd**

Pada kode di yang telah dibuat, kita menggunakan paket autograd dalam PyTorch, yang menyediakan diferensiasi otomatis untuk semua operasi pada Tensor. Ketika kita mendefinisikan Tensor dengan `requires_grad=True`, PyTorch akan melacak semua operasi yang dilakukan pada Tensor tersebut.

Pada contoh pertama, kita menghasilkan Tensor `x` dengan bentuk (3,) dan `y` sebagai hasil operasi penambahan Tensor `x` dengan skalar 2. Tensor `y` memiliki atribut `grad_fn` yang merujuk pada fungsi yang telah menciptakan Tensor tersebut.

Selanjutnya, kita melakukan lebih banyak operasi pada Tensor `y`, yaitu mengalikan `y` dengan dirinya sendiri dan mengambil rata-rata dari hasil perkalian tersebut. Kemudian, kita melakukan backpropagation dengan memanggil fungsi `.backward()` pada Tensor `z`. Setelah itu, kita dapat melihat nilai gradien dari Tensor `x` dengan menggunakan atribut `.grad` dari Tensor tersebut.

Selanjutnya, kita memperlihatkan contoh dengan Tensor non-skalar. Kita mendefinisikan Tensor `x` dengan bentuk (3,) dan melakukan operasi perkalian berulang kali pada Tensor `y`. Kemudian, kita mendefinisikan Tensor `v` dengan bentuk yang sesuai dan menggunakan metode `.backward()` dengan argumen `v` untuk menghitung gradien. Hasil gradien dari Tensor `x` dapat dilihat dengan menggunakan atribut `.grad` dari Tensor tersebut.

Kode selanjutnya menunjukkan cara menghentikan Tensor untuk melacak riwayat. Terdapat beberapa metode yang dapat digunakan, seperti `.requires_grad_(False)`, `.detach()`, atau `torch.no_grad()`. Ketiga metode tersebut menghasilkan Tensor baru yang memiliki konten yang sama namun tidak melacak riwayat. Hal ini berguna saat kita ingin mengupdate bobot pada loop pelatihan, di mana operasi pembaruan bobot tidak perlu dicatat dalam komputasi gradien.

Terakhir, kita melihat contoh penggunaan `.zero_()` untuk mengosongkan gradien sebelum langkah optimisasi baru. Pada contoh tersebut, kita mengoptimasi bobot model dengan mengurangi gradien bobot dengan laju pembelajaran dan mengatur gradien bobot ke nol setiap langkah optimisasi.

- **Backpropagation & Gradient Descent**

Pada kode yang telah dibuat, kami mengilustrasikan proses optimisasi parameter menggunakan PyTorch. Tujuan dari optimasi ini adalah untuk menemukan nilai parameter yang menghasilkan hasil prediksi yang paling dekat dengan nilai yang diharapkan.

Langkah-langkah dalam proses optimisasi adalah sebagai berikut:

**1. Inisialisasi Parameter:** Kami mendefinisikan parameter `w` dengan nilai awal 1.0 dan mengatur `requires_grad=True` agar PyTorch melacak gradien parameter ini selama proses optimisasi.

**2. Forward Pass:** Kami melakukan forward pass untuk menghitung hasil prediksi `y_predicted`. Hal ini dilakukan dengan mengalikan nilai parameter `w` dengan nilai input `x`.

**3. Perhitungan Loss:** Kami menghitung loss, yaitu selisih antara hasil prediksi `y_predicted` dengan nilai yang diharapkan `y`. Pada contoh ini, kami menggunakan mean squared error (MSE) sebagai fungsi loss.

**4. Backward Pass:** Kami melakukan backward pass dengan memanggil metode `backward()` pada loss. Ini akan menghitung gradien loss terhadap parameter `w` menggunakan aturan rantai.

**5. Perubahan Parameter:** Kami melakukan pembaruan parameter menggunakan metode penurunan gradien (gradient descent). Dalam contoh ini, kami menggunakan laju pembelajaran (learning rate) sebesar 0.01. Kami mengurangi nilai parameter `w` dengan hasil perkalian laju pembelajaran dengan gradien parameter `w`.

**6. Pembersihan Gradien:** Sebelum melakukan iterasi berikutnya, kami mengosongkan gradien parameter `w` menggunakan metode `zero_()` agar gradien tidak terakumulasi dari iterasi sebelumnya.

Langkah-langkah 2-6 kemudian dapat diulang secara iteratif untuk melakukan optimasi lebih lanjut. Pada setiap iterasi, kami melakukan forward pass untuk menghitung loss, backward pass untuk menghitung gradien, pembaruan parameter, dan pembersihan gradien sebelum iterasi berikutnya.

Metode optimasi ini bertujuan untuk mencari nilai parameter yang meminimalkan loss dan menghasilkan hasil prediksi yang lebih baik seiring dengan iterasi yang dilakukan.

- **Model Loos & Optimizer**

Pada kode yang telah dibuat, saya mengilustrasikan proses pelatihan model regresi linear menggunakan PyTorch. Model regresi linear digunakan untuk mempelajari hubungan linier antara input `X` dan output `Y`. Tujuan dari pelatihan ini adalah untuk menemukan parameter `w` yang paling sesuai untuk memodelkan hubungan tersebut.

Langkah-langkah dalam proses pelatihan adalah sebagai berikut:

**1. Desain Model:** Kami mendefinisikan model regresi linear dengan menggunakan kelas `nn.Linear` dari PyTorch. Model ini menerima input dengan dimensi `input_size` (jumlah fitur) dan menghasilkan output dengan dimensi `output_size` (jumlah fitur). Model ini akan mengimplementasikan forward pass, yang akan menghitung prediksi output berdasarkan input.

**2. Definisikan Loss dan Optimizer:** Kami menggunakan Mean Squared Error (MSE) sebagai fungsi loss untuk mengukur selisih antara prediksi model dan nilai target. Kami menggunakan optimizer Stochastic Gradient Descent (SGD) untuk mengoptimalkan parameter model. Kami mengatur laju pembelajaran (learning rate) sebesar 0.01.

**3. Training Loop:** Melakukan iterasi sebanyak `n_iters` kali untuk melatih model.

Pada setiap iterasi yang kelipatan 10, kami mencetak nilai parameter `w` dan loss untuk melihat perkembangan pelatihan.

Setelah selesai melakukan pelatihan, kami mencetak hasil prediksi dari model dengan memberikan input `X_test`.

Metode ini bertujuan untuk menemukan parameter `w` yang menghasilkan prediksi yang paling dekat dengan nilai target. Semakin banyak iterasi yang dilakukan, semakin baik model akan belajar menyesuaikan parameter untuk meminimalkan loss dan menghasilkan prediksi yang lebih baik.

- **Regresi Linear**

Pada kode yang telah dibuat, kami menggunakan PyTorch untuk melatih model regresi linear pada dataset yang disiapkan. Dataset yang digunakan dibuat menggunakan fungsi `make_regression` dari library scikit-learn. Dataset ini terdiri dari 100 sampel dengan satu fitur dan disertai dengan nilai target yang ditambahi dengan noise untuk membuatnya lebih realistis.

Setelah mempersiapkan dataset, kami membangun model regresi linear menggunakan modul `nn.Linear` dari PyTorch. Model ini memiliki satu layer linier dengan input size yang sesuai dengan jumlah fitur dan output size yang sebesar 1.

Kemudian, kami mendefinisikan fungsi loss yang akan digunakan, yaitu *Mean Squared Error (MSE)*, dan optimizer *Stochastic Gradient Descent (SGD)*. Laju pembelajaran (learning rate) ditetapkan sebagai 0.01.

Dalam loop pelatihan sebanyak 100 epoch, kami melakukan langkah-langkah berikut:

- Melakukan forward pass dengan memasukkan input `X` ke dalam model untuk menghasilkan prediksi `y_predicted`.

- Menghitung loss antara prediksi `y_predicted` dan nilai target `y` menggunakan fungsi loss MSE.

- Melakukan backward pass untuk menghitung gradien loss terhadap parameter model.

- Menggunakan optimizer untuk memperbarui parameter model berdasarkan gradien yang dihitung.

- Mengosongkan gradien menggunakan `zero_grad()` pada optimizer untuk langkah selanjutnya.

Selama pelatihan, setiap 10 epoch, kami mencetak nilai loss untuk melihat perkembangan pelatihan.

Setelah pelatihan selesai, kami melakukan prediksi menggunakan model pada seluruh data `X` dan memplot hasilnya menggunakan matplotlib. Data asli ditandai dengan titik merah, sedangkan garis biru adalah hasil prediksi model.

Metode ini bertujuan untuk menemukan parameter yang paling baik untuk memodelkan hubungan linier antara input dan output pada dataset yang diberikan. Melalui iterasi pelatihan, model akan menyesuaikan parameter untuk meminimalkan kesalahan prediksi dan menghasilkan garis regresi yang cocok dengan data.

- **Regresi Logistik**

Pada kode yang telah dibuat, kami menggunakan PyTorch untuk melatih model pada dataset Breast Cancer Wisconsin. Kami membagi dataset menjadi data pelatihan (80%) dan data pengujian (20%) menggunakan train_test_split dari scikit-learn. Selanjutnya, kami melakukan penskalaan data menggunakan StandardScaler untuk memastikan bahwa semua fitur memiliki skala yang serupa.

Setelah itu, kami mendefinisikan model dengan menggunakan class Model yang merupakan subclass dari nn.Module. Model ini adalah model regresi logistik yang memiliki satu layer linier diikuti oleh fungsi aktivasi sigmoid untuk menghasilkan probabilitas kelas. Parameter model diperbarui selama pelatihan menggunakan optimizer torch.optim.SGD dengan laju pembelajaran (learning rate) sebesar 0.01.

Dalam loop pelatihan, kami melakukan langkah-langkah berikut:

- Melakukan forward pass dengan memasukkan data pelatihan X_train ke dalam model untuk menghasilkan prediksi y_pred.

- Menghitung loss antara prediksi y_pred dan label pelatihan y_train menggunakan fungsi loss nn.BCELoss yang cocok untuk tugas klasifikasi biner.

- Melakukan backward pass untuk menghitung gradien loss terhadap parameter model.

- Menggunakan optimizer untuk memperbarui parameter model berdasarkan gradien yang dihitung.

- Mengosongkan gradien menggunakan zero_grad() pada optimizer untuk langkah selanjutnya.

Selama pelatihan, setiap 10 epoch, kami mencetak nilai loss untuk melihat perkembangan pelatihan.

Setelah pelatihan selesai, kami melakukan prediksi pada data pengujian X_test menggunakan model yang telah dilatih. Kami menghitung akurasi prediksi dengan membandingkan prediksi yang dibulatkan menjadi kelas biner dengan label pengujian y_test. Akurasi dihitung sebagai jumlah prediksi yang benar dibagi dengan jumlah total sampel pengujian.

- **Dataset & Dataloader**

Pada kode yang telah dibuat, kami menggunakan PyTorch untuk mengelola pelatihan model menggunakan DataLoader. DataLoader adalah utilitas yang membantu dalam membagi dataset menjadi batch-batch kecil dan mengelola proses loading data secara efisien.

Untuk mendemonstrasikan penggunaan DataLoader, kami mengimplementasikan sebuah kelas `WineDataset` yang merupakan subclass dari `torch.utils.data.Dataset`. Kelas ini bertujuan untuk membaca data dari file CSV menggunakan NumPy, dan mengubahnya menjadi tensor menggunakan PyTorch.

Dalam kelas `WineDataset`, kami mengimplementasikan tiga metode utama: `__init__`, `__getitem__`, dan `__len__`. Metode `__init__` digunakan untuk

menginisialisasi dataset dan membaca data dari file CSV. Metode `__getitem__` digunakan untuk mengambil satu sampel data dengan indeks tertentu. Metode `__len__` mengembalikan jumlah total sampel dalam dataset.

Setelah kita memiliki dataset, kita menggunakan DataLoader untuk memuat data dengan batch. Kami membuat objek `train_loader` menggunakan DataLoader, yang mengambil dataset yang telah kita definisikan sebelumnya. Dalam contoh ini, kami menggunakan batch_size=4, sehingga DataLoader akan membagi dataset menjadi batch-batch dengan ukuran 4.

Kita dapat mengakses batch-batch data menggunakan iterator dari DataLoader. Pada contoh ini, kita menggunakan `iter(train_loader)` untuk mengubah DataLoader menjadi iterator, dan kemudian menggunakan `next(dataiter)` untuk mendapatkan satu batch data. Ini memungkinkan kita untuk melihat dan mengakses data secara batch.

Selanjutnya, kita melihat contoh penggunaan DataLoader dengan dataset MNIST dari torchvision.datasets. Dalam contoh ini, kita menggunakan MNIST dataset, yang tersedia dalam torchvision. Kami menggunakan DataLoader untuk memuat data dari dataset MNIST dengan batch_size=3.

Pada akhirnya, kita menggunakan `iter(train_loader)` untuk mengakses iterator dari DataLoader dan menggunakan `next(dataiter)` untuk mendapatkan satu batch data. Kita mencetak bentuk dari data input dan label untuk memverifikasi bahwa batch data telah dimuat dengan benar.

Dengan menggunakan DataLoader, kita dapat mengelola proses pelatihan model secara efisien dengan membagi dataset menjadi batch-batch kecil dan memuatnya dengan iterasi.

- **Transformasi Dataset**

Dalam kode yang telah dibuat, kami memperkenalkan konsep transformasi pada dataset menggunakan PyTorch. Transformasi memungkinkan kita untuk menerapkan operasi khusus pada data sebelum memasukkannya ke dalam model.

Pertama, kami mendefinisikan kelas `WineDataset` yang sama seperti sebelumnya, tetapi kali ini kami menambahkan argumen `transform` pada konstruktor. Argumen ini digunakan untuk menerima objek transformasi yang akan diterapkan pada setiap sampel dataset.

Selanjutnya, kami mendefinisikan dua transformasi khusus, yaitu `ToTensor` dan `MulTransform`. Kelas `ToTensor` digunakan untuk mengubah numpy.ndarray menjadi tensor PyTorch. Kelas `MulTransform` mengalikan input dengan faktor tertentu.

Pada contoh pertama, kami mencetak dataset tanpa transformasi. Hasilnya, tipe data features dan labels adalah numpy.ndarray.

Pada contoh kedua, kami menerapkan transformasi `ToTensor()` pada dataset. Hasilnya, tipe data features dan labels menjadi torch.Tensor.

Pada contoh ketiga, kami menggunakan transformasi gabungan dengan menggunakan `Compose`. Kami menggabungkan transformasi `ToTensor()` dan `MulTransform(4)`. Hasilnya, fitur (features) dikalikan dengan faktor 4 setelah diubah menjadi tensor.

Dengan menggunakan transformasi, kita dapat dengan mudah memodifikasi data sebelum memasukkannya ke dalam model. Ini sangat berguna untuk pra pemrosesan data, normalisasi, augmentasi data, dan operasi lainnya yang dapat meningkatkan performa model.

- **Softmax dan Crossentropy**

Dalam kode yang telah dibuat, kami memperkenalkan beberapa konsep dasar terkait fungsi softmax, cross-entropy loss, dan penggunaannya dalam PyTorch.

Fungsi softmax digunakan untuk mengkonversi skor (logits) menjadi probabilitas yang dinormalisasi. Pada contoh pertama, kami mengimplementasikan fungsi softmax secara manual menggunakan NumPy. Pada contoh kedua, kami menggunakan fungsi softmax bawaan di PyTorch (`torch.softmax`).

Kemudian, kami mendefinisikan fungsi cross-entropy yang digunakan untuk mengukur performa model klasifikasi. Pada contoh pertama, kami mengimplementasikan fungsi cross-entropy secara manual menggunakan NumPy. Pada contoh kedua, kami menggunakan kelas `nn.CrossEntropyLoss` di PyTorch.

Setelah itu, kami menunjukkan penggunaan `nn.CrossEntropyLoss` dalam beberapa skenario. Pada contoh pertama, kami menggunakan tensor 1D untuk target dan input, mewakili kasus klasifikasi biner. Pada contoh kedua, kami menggunakan tensor 2D untuk target dan input, mewakili kasus klasifikasi multi kelas.

Terakhir, kami menunjukkan penggunaan `nn.BCELoss` untuk kasus klasifikasi biner dengan fungsi aktivasi sigmoid. Kami juga menunjukkan penggunaan `nn.CrossEntropyLoss` untuk kasus klasifikasi multi kelas tanpa fungsi aktivasi softmax pada lapisan output.

Dalam setiap contoh, kami mendefinisikan arsitektur model dan kriterium loss yang sesuai dengan kasus yang ada.

Pemahaman tentang softmax, cross-entropy loss, dan penggunaannya dalam PyTorch penting untuk memahami dan mengimplementasikan algoritma klasifikasi dengan baik.

- **Fungsi Aktivasi**

Dalam kode yang telah dibuat, kami memperkenalkan beberapa fungsi aktivasi yang umum digunakan dalam jaringan saraf, serta cara penggunaannya dalam PyTorch.

Fungsi aktivasi digunakan untuk memperkenalkan sifat non-linearitas ke dalam model jaringan saraf. Beberapa fungsi aktivasi yang umum digunakan adalah:

- Softmax: Fungsi softmax mengonversi skor (logits) menjadi probabilitas yang dinormalisasi. PyTorch menyediakan fungsi softmax sebagai `torch.softmax` atau sebagai objek `nn.Softmax`.

- Sigmoid: Fungsi sigmoid mengonversi nilai menjadi rentang antara 0 dan 1, yang biasanya digunakan dalam masalah klasifikasi biner. PyTorch menyediakan fungsi sigmoid sebagai `torch.sigmoid` atau sebagai objek `nn.Sigmoid`.

- Tanh: Fungsi tanh (tangens hiperbolik) mengonversi nilai menjadi rentang antara -1 dan 1. PyTorch menyediakan fungsi tanh sebagai `torch.tanh` atau sebagai objek `nn.Tanh`.

- ReLU (Rectified Linear Unit): Fungsi ReLU menggantikan nilai negatif dengan nol dan mempertahankan nilai positif. PyTorch menyediakan fungsi ReLU sebagai `torch.relu` atau sebagai objek `nn.ReLU`.

- Leaky ReLU: Fungsi Leaky ReLU mirip dengan ReLU, tetapi mengizinkan nilai negatif yang sangat kecil (leaky) untuk meloloskan sedikit informasi. PyTorch menyediakan fungsi Leaky ReLU sebagai `F.leaky_relu` atau sebagai objek `nn.LeakyReLU`.

Kita dapat menggunakan fungsi-fungsi ini secara langsung dalam forward pass model jaringan saraf. Pada contoh terakhir dalam kode yang telah dibuat, ada dua opsi yang ditunjukkan: menggunakan objek `nn.Module` untuk mendefinisikan fungsi aktivasi sebagai atribut dan memanggilnya dalam forward pass, atau menggunakan fungsi aktivasi langsung dalam forward pass tanpa menggunakan objek `nn.Module`.

- **Feed Forward Net**

Kode yang telah dibuat adalah contoh implementasi jaringan saraf berbasis fully connected (FC) dengan satu lapisan tersembunyi (hidden layer) menggunakan PyTorch untuk melakukan klasifikasi pada dataset MNIST.

Dataset MNIST adalah dataset gambar digit tulisan tangan yang terdiri dari gambar-gambar 28x28 piksel dari digit 0 hingga 9. Kode yang telah dibuat menggunakan

modul `torchvision.datasets` untuk mengunduh dan memuat dataset MNIST. Dataset ini dibagi menjadi dua bagian: data latih (train_dataset) dan data uji (test_dataset).

Selanjutnya, kode menggunakan objek `torch.utils.data.DataLoader` untuk membuat loader data yang memungkinkan pengulangan (iteration) mudah melalui dataset dalam batch-batch ukuran tertentu. DataLoader akan membagi dataset menjadi batch-batch kecil yang dapat digunakan untuk melatih model secara efisien.

Kemudian, kode mendefinisikan struktur model jaringan saraf menggunakan kelas `NeuralNet`. Model ini memiliki tiga lapisan: lapisan input, lapisan tersembunyi dengan fungsi aktivasi ReLU, dan lapisan output yang menghasilkan skor untuk masing-masing kelas digit. Model juga didefinisikan untuk menggunakan perangkat GPU (jika tersedia) untuk komputasi dengan menggunakan `.to(device)`.

Selanjutnya, kode mendefinisikan fungsi loss (CrossEntropyLoss) dan optimizer (Adam) yang digunakan untuk melatih model. Dalam pelatihan model, setiap batch dari data latih melewati proses forward pass, perhitungan loss, backward pass, dan pengoptimalan parameter menggunakan metode optimizer yang telah ditentukan.

Setelah melatih model, kode mengevaluasi kinerja model pada data uji. Model dijalankan pada mode evaluasi dengan menggunakan `torch.no_grad()` untuk menghindari perhitungan gradien yang tidak perlu. Model melakukan prediksi pada data uji dan menghitung akurasi dengan membandingkan prediksi dengan label yang benar.

- **CNN**

Kode yang telah dibuat adalah contoh implementasi jaringan saraf konvolusi (Convolutional Neural Network/CNN) menggunakan PyTorch untuk melakukan klasifikasi pada dataset CIFAR-10.

Dataset CIFAR-10 berisi gambar-gambar berukuran 32x32 piksel dalam 10 kelas yang berbeda seperti pesawat, mobil, burung, kucing, dan lain-lain. Kode yang telah dibuat menggunakan modul `torchvision.datasets` untuk mengunduh dan memuat dataset CIFAR-10. Dataset ini dibagi menjadi dua bagian: data latih (train_dataset) dan data uji (test_dataset).

Selanjutnya, kode mendefinisikan struktur model jaringan saraf menggunakan kelas `ConvNet`. Model ini terdiri dari beberapa lapisan konvolusi (Conv2d) dan pengurangan dimensi (MaxPool2d), diikuti oleh lapisan-lapisan linear (Linear) untuk melakukan klasifikasi. Model menggunakan fungsi aktivasi ReLU untuk setiap lapisan konvolusi dan lapisan linear.

Setelah mendefinisikan model, kode mendefinisikan fungsi loss (CrossEntropyLoss) dan optimizer (SGD) yang digunakan untuk melatih model. Dalam pelatihan model, setiap

batch dari data latih melewati proses forward pass, perhitungan loss, backward pass, dan pengoptimalan parameter menggunakan metode optimizer yang telah ditentukan.

Setelah melatih model selama beberapa epoch, kode mengevaluasi kinerja model pada data uji. Model dijalankan pada mode evaluasi dengan menggunakan `torch.no_grad()` untuk menghindari perhitungan gradien yang tidak perlu. Model melakukan prediksi pada data uji dan menghitung akurasi secara keseluruhan dan akurasi untuk setiap kelas.

- **Transfer Learning**

Kode yang telah dibuat adalah contoh implementasi transfer learning dengan menggunakan model ResNet-18 pada dataset Hymenoptera. Dataset ini berisi gambar-gambar semut dan lebah yang digunakan untuk melatih model dalam tugas klasifikasi.

Pertama, kode mengekstrak dataset dari file zip menggunakan `zipfile.ZipFile` dan `extractall` untuk mengekstrak file zip ke direktori tujuan.

Selanjutnya, kode mendefinisikan transformasi data menggunakan `transforms.Compose` untuk melakukan pra-pemrosesan gambar. Terdapat dua jenis transformasi yang didefinisikan: transformasi untuk data latih (`'train'`) dan transformasi untuk data validasi (`'val'`). Transformasi tersebut mencakup operasi seperti pembesaran acak, pemotongan acak, normalisasi, dan lain-lain.

Setelah itu, kode memuat dataset menggunakan `datasets.ImageFolder`. Dataset ini dibagi menjadi dua bagian: data latih (`'train'`) dan data validasi (`'val'`). Data tersebut kemudian dimuat ke dalam `torch.utils.data.DataLoader` untuk mengatur pengumpulan batch data dengan ukuran 4, pengacakan, dan penanganan beban kerja (num_workers).

Kemudian, model ResNet-18 dari `torchvision.models` dimuat dengan parameter `pretrained=True`, yang berarti model akan menggunakan bobot yang telah dilatih sebelumnya pada ImageNet. Model tersebut dimodifikasi dengan mengubah lapisan terakhirnya menjadi `nn.Linear(num_ftrs, 2)` untuk menyesuaikan jumlah kelas dalam dataset Hymenoptera.

Setelah itu, fungsi `train_model` didefinisikan untuk melatih model. Fungsi ini menerima model, kriteria loss, optimizer, scheduler, dan jumlah epoch sebagai argumen. Dalam setiap epoch, fungsi akan melatih model pada data latih dan melakukan evaluasi pada data validasi. Fungsi ini mencetak loss dan akurasi pada setiap fase (train dan val) untuk setiap epoch.

Setelah melatih model dengan transfer learning, kode mendemonstrasikan dua pendekatan transfer learning yang berbeda:

**1. Fine-tuning ConvNet:** Model ResNet-18 dimuat dengan bobot yang telah dilatih sebelumnya, dan lapisan terakhirnya diubah untuk klasifikasi dua kelas. Seluruh model, termasuk lapisan terakhir, dioptimalkan selama pelatihan.

**2. ConvNet as fixed feature extractor:** Model ResNet-18 dimuat dengan bobot yang telah dilatih sebelumnya, dan semua parameter kecuali lapisan terakhirnya dibekukan (requires_grad=False). Hanya lapisan terakhir yang akan dioptimalkan selama pelatihan.

Kedua pendekatan tersebut menggunakan kriteria loss CrossEntropyLoss dan optimizer SGD. Mereka juga menggunakan scheduler lr_scheduler.StepLR untuk mengatur penurunan learning rate setelah setiap jumlah epoch tertentu.

- **TensorBoard**

Kode yang telah diberikan adalah contoh dari pelatihan model jaringan saraf pada dataset MNIST menggunakan PyTorch dan visualisasi proses pelatihan menggunakan TensorBoard.

Pertama, impor semua pustaka yang diperlukan, termasuk PyTorch, torch.nn, torchvision, transforms, dan matplotlib.pyplot. Selanjutnya, kita dapat mengimpor SummaryWriter dari torch.utils.tensorboard untuk mengaktifkan TensorBoard.

Setelah itu, konfigurasi perangkat dipilih berdasarkan ketersediaan GPU. Hyperparameter seperti ukuran input, ukuran tersembunyi, jumlah kelas, jumlah epoch, ukuran batch, dan tingkat pembelajaran didefinisikan.

Kemudian, dataset MNIST diunduh dan dibagi menjadi dataset pelatihan dan pengujian. Data loader dibuat untuk memuat data dalam batch selama pelatihan dan evaluasi.

Selanjutnya, beberapa contoh data dari dataset pengujian ditampilkan menggunakan matplotlib.pyplot dan diatur dalam grid. Grid ini ditambahkan ke TensorBoard menggunakan writer.add_image.

Model jaringan saraf yang terdiri dari satu lapisan tersembunyi diimplementasikan menggunakan kelas NeuralNet. Fungsi forward menghubungkan lapisan-lapisan dan menghasilkan keluaran tanpa aktivasi dan tanpa softmax di akhirnya.

Pada tahap ini, fungsi loss dan optimizer didefinisikan. Kriteria CrossEntropyLoss digunakan sebagai fungsi loss, dan optimizer Adam digunakan untuk mengoptimalkan parameter model.

TensorBoard digunakan untuk menambahkan grafik representasi model menggunakan writer.add_graph. Hal ini membantu memvisualisasikan struktur model di TensorBoard.

Setelah itu, model dilatih menggunakan data pelatihan. Selama setiap epoch, iterasi dilakukan pada data loader pelatihan. Untuk setiap iterasi, output model diperoleh dengan meneruskan gambar ke model, dan loss dihitung menggunakan kriteria CrossEntropyLoss. Optimizer digunakan untuk memperbarui parameter model berdasarkan loss. Selama pelatihan, running loss dan running accuracy dihitung dan ditambahkan ke TensorBoard menggunakan writer.add_scalar.

Setelah pelatihan, model dievaluasi menggunakan data pengujian. Output model untuk setiap gambar pengujian diperoleh, dan akurasi dihitung dengan membandingkan label yang diprediksi dengan label sebenarnya. Hasil akurasi juga ditambahkan ke TensorBoard.

Terakhir, kurva Precision-Recall (PR) ditambahkan ke TensorBoard untuk setiap kelas menggunakan writer.add_pr_curve. Ini membantu memvisualisasikan performa model dalam mengenali kelas-kelas tertentu.

- **Save & Load Modules**

**Metode Pertama: Menyimpan dan Memuat Seluruh Model**

Metode ini melibatkan penyimpanan dan pemulihan seluruh model beserta semua parameter dan strukturnya. kita dapat menyimpan model dengan menggunakan torch.save(model, PATH) dan memuatnya dengan model = torch.load(PATH). Setelah memuat model, kita dapat menggunakannya untuk inferensi dengan memanggil model.eval().

**Metode Kedua: Menyimpan dan Memuat State Dictionary**

Metode ini lebih disarankan karena hanya menyimpan state dictionary dari model. kita dapat menyimpan state dictionary dengan torch.save(model.state_dict(), PATH) dan memuatnya dengan model.load_state_dict(torch.load(PATH)). kita perlu membuat model baru terlebih dahulu dengan menggunakan struktur yang sama, dan kemudian memuat state dictionary ke model tersebut. Setelah memuat state dictionary, kita juga perlu memanggil model.eval() untuk inferensi.

**Metode Ketiga: Menyimpan dan Memuat Checkpoint**

Metode ini berguna saat kita ingin menyimpan lebih dari sekadar state dictionary model. Misalnya, kita ingin menyimpan juga optimizer dan informasi lainnya. kita dapat membuat checkpoint yang berisi informasi seperti epoch, state dictionary model, dan state dictionary optimizer, dan menyimpannya dengan torch.save(checkpoint, FILE). Kemudian, untuk memuat checkpoint, kita perlu membuat model baru dengan struktur yang sama dan memuat state dictionary model dan state dictionary optimizer dari checkpoint dengan model.load_state_dict(checkpoint['model_state']) dan optimizer.load_state_dict(checkpoint['optim_state']).

Ketika kita menyimpan dan memuat model, penting untuk memperhatikan perangkat yang digunakan. kita dapat menggunakan metode model.to(device) untuk memindahkan model ke perangkat yang diinginkan sebelum menyimpan atau memuatnya. kita juga dapat

menggunakan argumen map_location saat memuat model untuk menentukan perangkat yang digunakan. Pastikan untuk memanggil model.eval() setelah memuat model untuk mengatur layer dropout dan batch normalization dalam mode evaluasi.

Secara keseluruhan, kode tersebut memberikan contoh tentang cara menyimpan dan memuat model dalam PyTorch dengan menggunakan seluruh model, state dictionary, atau checkpoint, serta memberikan panduan untuk memindahkan model antara perangkat CPU dan GPU saat menyimpan dan memuatnya.

- **Kesimpulan**

Dalam laporan ini, kita akan membahas konsep dasar Deep Learning menggunakan PyTorch, kerangka kerja populer untuk pemrosesan data dan pembelajaran mendalam. Fokus utama kita akan pada topik tensor, autograd, backpropagation, dan optimisasi.

Tensor adalah struktur data fundamental dalam PyTorch, digunakan untuk mewakili data multidimensional. Kita dapat membuat tensor dengan fungsi `torch.empty()`, `torch.rand()`, `torch.zeros()`, dan `torch.ones()`. Operasi tensor seperti penambahan, pengurangan, perkalian, dan pembagian elemen demi elemen juga tersedia.

Autograd adalah fitur PyTorch yang menyediakan diferensiasi otomatis. Saat kita mendefinisikan tensor dengan `requires_grad=True`, PyTorch akan melacak operasi pada tensor tersebut. Ini memungkinkan kita untuk melakukan backpropagation dan menghitung gradien dengan mudah menggunakan metode `.backward()`.

Backpropagation digunakan dalam deep learning untuk menghitung gradien loss terhadap parameter model. PyTorch secara otomatis mengelola backpropagation melalui autograd. Dengan pembaruan parameter menggunakan optimisasi seperti gradient descent, kita dapat mengoptimalkan model untuk menghasilkan prediksi yang lebih baik.

Selain itu, laporan ini juga membahas penggunaan tensor dalam integrasi dengan NumPy untuk konversi data, serta kemampuan PyTorch untuk memanfaatkan komputasi GPU untuk kecepatan pemrosesan yang lebih tinggi.

Dengan pemahaman tentang tensor, autograd, backpropagation, dan optimisasi, kita dapat mengembangkan dan melatih model deep learning dengan efisien menggunakan PyTorch.