Chapter 11: Melatih Jaringan Saraf Dalam (Training Deep Neural Networks)

Pendahuluan

Meskipun secara teoretis Jaringan Saraf Dalam (*Deep Neural Networks* - DNN) mampu memodelkan fungsi yang sangat kompleks, proses pelatihannya penuh dengan tantangan. Menumpuk banyak lapisan dapat menyebabkan masalah serius seperti gradien yang tidak stabil, yang membuat pelatihan menjadi sangat lambat atau bahkan gagal total. Chapter ini membahas secara mendalam berbagai masalah yang muncul saat melatih DNN dan menyajikan serangkaian teknik dan solusi canggih untuk mengatasinya, yang telah menjadi praktik standar dalam Deep Learning modern.

1. Masalah Gradien yang Lenyap/Meledak (The Vanishing/Exploding Gradients Problem)

Masalah ini adalah kendala utama dalam melatih DNN. Selama proses *backpropagation*, gradien seringkali menjadi semakin kecil saat disebarkan dari lapisan output ke lapisan input. Akibatnya, bobot dari lapisan-lapisan awal hampir tidak diperbarui, dan pelatihan tidak pernah konvergen ke solusi yang baik. Ini disebut **Vanishing Gradients**. Sebaliknya, dalam beberapa kasus, gradien juga bisa menjadi semakin besar, menyebabkan pembaruan bobot yang sangat besar dan membuat model menjadi tidak stabil. Ini disebut **Exploding Gradients**.

Penyebabnya adalah kombinasi dari fungsi aktivasi tradisional seperti fungsi logistik (sigmoid) dan metode inisialisasi bobot standar (misalnya, distribusi normal acak), yang secara kumulatif menyebabkan varians sinyal menjadi jauh lebih besar atau lebih kecil saat melewati setiap lapisan.

2. Solusi untuk Gradien yang Tidak Stabil

2.1 Inisialisasi Bobot (Weight Initialization)

Salah satu solusi paling efektif adalah dengan menggunakan strategi inisialisasi bobot yang lebih cerdas. Tujuannya adalah untuk menjaga agar varians output dari setiap lapisan sama dengan varians inputnya, sehingga sinyal dapat mengalir dengan baik ke kedua arah (maju dan mundur).

- **Glorot (Xavier) Initialization**: Strategi ini menginisialisasi bobot koneksi secara acak dengan *mean* 0 dan varians spesifik: Var(W) = 1 / fan_avg, di mana fan_avg = (fan_in + fan_out) / 2. fan_in adalah jumlah input dan fan_out adalah jumlah output dari lapisan tersebut. Ini adalah default di Keras.
- **He Initialization**: Ini adalah varian dari inisialisasi Glorot yang dirancang khusus untuk fungsi aktivasi keluarga ReLU. Variansnya adalah Var(W) = 2 / fan_in. Metode ini sangat penting karena ReLU mematikan sekitar setengah dari neuron (yang bernilai negatif), sehingga mengubah dinamika varians.

2.2 Fungsi Aktivasi Non-Saturasi

Fungsi aktivasi seperti sigmoid dan tanh mengalami "saturasi" pada nilai ekstrim (mendekati 0 atau 1, dan -1 atau 1), di mana turunannya menjadi hampir nol. Ini menghentikan aliran gradien. Penggunaan fungsi aktivasi non-saturasi dapat mengatasi masalah ini.

- **ReLU (Rectified Linear Unit)**: Seperti dibahas sebelumnya, ReLU tidak mengalami saturasi untuk nilai positif. Namun, ia memiliki masalah "Dying ReLU", di mana neuron dapat "mati" jika inputnya selalu negatif selama pelatihan.
- **Leaky ReLU**: Variasi dari ReLU yang memungkinkan gradien kecil untuk input negatif, mencegah neuron mati. Rumusnya adalah: $\text{LeakyReLU}_a(z) = \max(\alpha z, z)$. α adalah *hyperparameter* yang menentukan tingkat "kebocoran".
- **ELU (Exponential Linear Unit)**: Fungsi ini berkinerja lebih baik dari varian ReLU lainnya. Ia dapat menghasilkan nilai negatif, yang memungkinkan rata-rata output lapisan lebih dekat ke nol, sehingga membantu mengurangi masalah gradien yang lenyap.
- **SELU (Scaled ELU)**: Varian khusus dari ELU yang, jika kondisi tertentu terpenuhi (arsitektur sekuensial, inisialisasi LeCun, dan input terstandarisasi), dapat membuat jaringan menjadi *self-normalizing*. Artinya, output dari setiap lapisan akan cenderung mempertahankan *mean* 0 dan standar deviasi 1 selama pelatihan, yang sangat efektif dalam mengatasi masalah gradien yang tidak stabil.

2.3 Batch Normalization (BN)

Batch Normalization adalah teknik yang sangat kuat yang mengatasi masalah gradien tidak stabil. BN menambahkan sebuah operasi pada setiap lapisan (biasanya sebelum fungsi aktivasi) yang menormalisasi inputnya.

Proses Batch Normalization:

1. **Normalisasi**: Algoritma menghitung *mean* dan standar deviasi dari input pada *mini-batch* saat ini, lalu menormalisasinya (mengurangi *mean* dan membagi dengan standar deviasi) sehingga hasilnya memiliki *mean* 0 dan varians 1.
2. **Scaling dan Shifting**: Setelah normalisasi, BN mengalikan hasilnya dengan parameter *scaling* (**γ**, gamma) dan menambahkan parameter *shifting* (**β**, beta). Kedua parameter ini dipelajari oleh model, memungkinkan jaringan untuk menentukan distribusi input yang optimal untuk lapisan berikutnya.

Manfaat Batch Normalization:

- Secara signifikan mengurangi masalah *vanishing gradients*.
- Memungkinkan penggunaan *learning rate* yang jauh lebih tinggi, mempercepat pelatihan.
- Bertindak sebagai regularizer, sehingga dapat mengurangi atau bahkan menghilangkan kebutuhan akan teknik regularisasi lain seperti Dropout.

2.4 Gradient Clipping

Teknik ini digunakan untuk mengatasi masalah **exploding gradients**. Idenya sangat sederhana: jika gradien melebihi *threshold* (batas) tertentu selama *backpropagation*, gradien tersebut akan "dipotong" atau diskalakan ulang agar tidak melebihi batas tersebut.

3. Menggunakan Kembali Layer Pre-trained (Transfer Learning)

Untuk masalah yang kompleks (misalnya, klasifikasi gambar), melatih DNN dari awal membutuhkan data yang sangat besar dan waktu yang lama. **Transfer Learning** adalah jalan pintas yang sangat efektif. Idenya adalah menemukan jaringan saraf yang sudah ada (disebut *pretrained model*) yang telah dilatih pada dataset besar dan serupa (misalnya, ImageNet), lalu menggunakan kembali lapisan-lapisan bawahnya.

Lapisan-lapisan bawah dari sebuah jaringan cenderung mempelajari fitur-fitur generik (seperti deteksi tepi, tekstur, bentuk), yang berguna untuk banyak tugas. Prosesnya adalah:

1. **Bekukan (Freeze)** bobot dari lapisan-lapisan yang digunakan kembali agar tidak berubah selama pelatihan.
2. Ganti lapisan output dari model pre-trained dengan lapisan output baru yang sesuai dengan tugas Anda.
3. Latih model pada lapisan baru ini saja.
4. (Opsional) Setelah beberapa *epoch*, **buka kembali (unfreeze)** beberapa lapisan atas yang dibekukan dan lanjutkan pelatihan dengan *learning rate* yang sangat kecil untuk menyempurnakan (*fine-tune*) model pada data spesifik Anda.

4. Optimizer yang Lebih Cepat

Selain Batch GD dan SGD, ada beberapa algoritma optimisasi yang lebih canggih yang konvergen jauh lebih cepat.

- **Momentum Optimization**: Mensimulasikan konsep momentum dari fisika. Algoritma ini tidak hanya menggunakan gradien saat ini untuk pembaruan, tetapi juga mengakumulasi gradien dari langkah-langkah sebelumnya. Ini membantunya melewati minimum lokal dan mempercepat konvergensi.
- **Nesterov Accelerated Gradient (NAG)**: Sedikit modifikasi dari Momentum. Ia menghitung gradien sedikit "di depan" dalam arah momentum, yang seringkali mempercepat konvergensi.
- **AdaGrad, RMSProp, dan Adam**: Ini adalah optimizer **adaptif**, yang berarti mereka menyesuaikan *learning rate* secara otomatis untuk setiap parameter.
    - **AdaGrad**: Memberikan *learning rate* yang lebih besar untuk parameter yang jarang diperbarui dan lebih kecil untuk yang sering diperbarui. Cenderung berhenti terlalu cepat.
    - **RMSProp**: Memperbaiki masalah AdaGrad dengan hanya mengakumulasi gradien dari iterasi-iterasi terbaru.
    - **Adam (Adaptive Moment Estimation)**: Optimizer yang paling populer saat ini. Adam menggabungkan ide dari Momentum dan RMSProp. Ia secara adaptif menyesuaikan *learning rate* sambil juga menyimpan jejak momentum.

5. Jadwal Laju Pembelajaran (Learning Rate Scheduling)

Menemukan *learning rate* yang baik bisa jadi sulit. Jika terlalu tinggi, model bisa divergen. Jika terlalu rendah, pelatihan akan sangat lambat. **Learning Rate Scheduling** adalah strategi untuk mengubah *learning rate* selama pelatihan. Pendekatan yang umum adalah memulai dengan *learning rate* yang relatif tinggi untuk konvergensi awal yang cepat, kemudian menurunkannya secara bertahap agar model dapat "tenang" dan menemukan solusi minimum yang lebih baik. Beberapa jadwal populer termasuk *power scheduling*, *exponential scheduling*, dan *performance scheduling* (mengurangi *learning rate* ketika *validation error* berhenti membaik).