CHAPTER 11

Training Deep Neural Networks

Chapter 11 explains how to **train deep neural networks** effectively by addressing vanishing/exploding gradients, improving optimization, leveraging pretraining/transfer learning, and regularizing large models.

Vanishing/exploding gradients and initialization

Deep nets can suffer from **vanishing gradients** (signals shrink as they backpropagate, making lower layers learn very slowly) or **exploding gradients** (numerical instability when gradients blow up). The chapter emphasizes three key mitigations:

- **Careful initialization**:
  - **Glorot/Xavier** initialization scales weights by $1/\sqrt{\text{fan\_avg}}$ for tanh/sigmoid, balancing variance across layers.
  - **He initialization** scales by $1/\sqrt{\text{fan\_in}/2}$ for ReLU-like activations, better preserving gradient flow.
- **Nonsaturating activations**: ReLU and variants (Leaky ReLU, PReLU, ELU, SELU) avoid the saturation regions of sigmoid/tanh where gradients are near zero; ELU/SELU can speed convergence further at the cost of more computation.
- **Batch Normalization**: normalizes intermediate activations per mini-batch, stabilizing distributions across layers, allowing higher learning rates, and acting as regularizer; implemented via BatchNormalization layers before/after activations.

Reusing pretrained layers and unsupervised pretraining

To cope with limited labeled data, deep nets often **reuse lower layers** learned on related tasks (**transfer learning**). Typical practice:

- Load a pretrained model, replace the output layer for the new task, **freeze** reused layers, and train the new head; then optionally **unfreeze** some top layers and fine-tune with a lower learning rate.
- The more similar the tasks and data, the more layers you can reuse; for very similar tasks, often only the output layer needs retraining.

When no suitable pretrained model exists but unlabeled data is abundant, **unsupervised or self-supervised pretraining** is an option. One trains an autoencoder or GAN (or other self-supervised objective) on raw inputs, then reuses the encoder or discriminator layers as initialization for the supervised task, fine-tuning them with labeled data. The chapter notes this was key to early deep learning (layer-wise pretraining) and remains useful when labeled data is scarce.

Faster optimizers

Beyond vanilla SGD, the chapter presents several **gradient-based optimizers** that speed convergence and handle ravines/plateaus better:

- **Momentum**: accumulates an exponentially decaying average of past gradients, giving updates an inertia that accelerates in consistent directions and damps oscillations; controlled by momentum coefficient (e.g., 0.9) and implemented via SGD(momentum=0.9).
- **Nesterov Accelerated Gradient**: computes gradients at the **lookahead** position (current params plus momentum step), often improving convergence; SGD(momentum=0.9, nesterov=True).
- **AdaGrad**: adapts learning rates per parameter using accumulated squared gradients, helping with sparse features but often decaying learning rates too aggressively, causing early stopping.
- **RMSProp**: like AdaGrad but with exponential decay of squared gradients, keeping learning rates from vanishing and becoming a strong default for many problems; RMSprop(lr=0.001, rho=0.9).
- **Adam**: combines momentum on gradients and RMSProp-style scaling using biased-corrected first and second moment estimates; usually works well with little tuning (defaults like Adam(lr≈1e-3, beta_1=0.9, beta_2=0.999)), and is widely used.
- **Nadam**: Adam plus Nesterov-style lookahead, sometimes offering small gains on certain tasks.

The chapter compares these optimizers and suggests SGD+momentum, RMSProp, or Adam/Nadam as practical choices, with learning rate still the most critical hyperparameter.

Learning rate schedules

A fixed learning rate is rarely optimal. The chapter introduces **learning rate scheduling** strategies:

- **Step decay**: reduce learning rate by a factor every few epochs.
- **Exponential decay**: $\eta_t = \eta_0 e^{-kt}$.
- **Performance-based**: monitor validation loss and reduce lr when progress stalls (ReduceLROnPlateau callback).
- **Warmup**: start with a small lr and ramp up, especially important for very large minibatches to avoid divergence.

A practical heuristic is to run a **learning rate range test**: sweep lr exponentially from very small to large in a short run, plot loss vs lr, and choose a value just below where loss starts exploding.

Regularization for deep nets

To combat overfitting in large networks, several **regularization** methods are recommended:

- **Weight decay (L2)** and **L1** regularization, applied via kernel_regularizer arguments or optimizer weight decay, encourage smaller or sparse weights.
- **Dropout**: randomly zeroes a fraction of activations during training, forcing redundancy and robustness; implemented with Dropout(rate) and typically used after dense layers or within conv blocks, with higher rates (e.g., 0.5) on fully connected layers.
- **MC Dropout**: keep dropout active at inference and run multiple passes to approximate Bayesian uncertainty from prediction variability.
- **Max-norm constraints**: clip weight vectors to a maximum norm per neuron after each update, limiting capacity and improving stability in some settings.

The chapter wraps up with **practical guidelines**: use ReLU-like activations, He initialization, BatchNorm, a good optimizer (Adam/RMSProp/SGD+momentum), appropriate learning rate scheduling, and regularization (dropout + weight decay), plus transfer learning whenever possible. For your Word summary, the key diagrams are: the vanishing/exploding gradient illustration, Glorot/He initialization comparison, BatchNorm schematic, momentum/Adam optimization curves, transfer learning/unsupervised pretraining workflows, and dropout vs no-dropout training behavior