Chapter 11: Training Deep Neural Networks

- The Vanishing/Exploding Gradients Problems
  - Problem
    - Gradients get smaller as progress to lower layers → lower layer weights don't change; or get too big
    - Unstable gradients: different layers learn at different speeds
    - Culprit: sigmoid activation and weight initialization using std. Normal; *saturates* top layers
    - When inputs large in abs value, sigmoid saturates at 0 or 1, derivative close to zero, so gradient doesn't change much, nothing left for lower layers
  - Glorot and He Initialization
    - Glorot: Want variance in to equal variance out (string of amplifiers analogy)
      - Can't guarantee if inputs != neurons
      - Compromise: initialize connection weights randomly where variance = 1/avg(fan in and fan out)
    - Variance changes with activation functions:
      - He: ReLU 2 /fan in
      - LeCun: SELU 1/fan in
  - Nonsaturating Activation Functions
    - Unstable gradients due in part to activation function
    - ReLU behaves better than sigmoid, does not saturate for +ve values
    - Dying ReLus: some neurons output nothing but zero
      - Neuron dies when wt'd sum of inputs are -ve for all instances in training set → outputs only zeros
    - Solutions:
      - Leaky ReLU: max(αz, z), α defines leakage, slope when z < 0
      - Randomized Leaky ReLU
      - Exponential LU: (α(exp(z)-1) if z <0, z if z >= 0
        - -ve when z < 0, vanishing gradients↓ avg output → 0
        - Nonzero gradient for z < 0 avoids dead neurons
        - If α = 1, function smooth everywhere → speeds up Gradient Descent
        - Slow to compute
      - Scaled ELU: stack of dense layers, all use SELU → self-normalization
        - All input features must be standardized
        - Initial weights with LeCun normal
        - Architecture must be sequential
      - Which activation function?
        - SELU > ELU > Leaky ReLU > ReLU > tanh > sigmoid
        - Architecture → no self-normalization → ELU
        - Run time latency → leaky ReLu
  - Batch Normalization

        - Zero-center and normalize input before or after activation, then scale and shift
        - Model learns optimal scale and mean of each layer's inputs
        - Calculates mean and stdev for each batch
        - Output = gamma times rescaled inputs + shift
        - Improves NN performance, reduces vanishing gradient such that can use sigmoid activation, low sensitivity to initial weights, use larger learning rates
        - Adds complexity, removes need to normalize input data
        - Slower predictions due to extra computations
        -
    - Gradient Clipping
- Reusing Pretrained Layers
    - Transfer Learning with Keras
    - Unsupervised Pretraining
    - Pretraining on an Auxiliary Task
- Faster Optimizers
    - Momentum Optimization
    - Nesterov Accelerated Gradient
    - AdaGrad
    - RMSProp
    - Adam and Nadam Optimization
    - Learning Rate Scheduling
        - Power: $\eta(t) = \eta_0 / (1 + t/s)^c$, c = 1 usually
            - Learning rate drops after s steps to η/2, η/3
            - Learning rate drops fast and then slow
        - Exponential: $\eta(t) = \eta_0 \, 0.1^{(t/s)}$
            - Drops by a factor of 10 every s steps
        - Piecewise constant: constant rate for number of epochs, then drop and use for another number of epochs
        - Performance: measure validation error every N steps. Reduce learning rate by factor of γ when error stops dropping
        - 1cycle: Grows linearly up to η_1 then decreases to η_0, with last few epochs dropping rate several orders of magnitude
        - Saving model → optimizer and learning rate get saved too
            - If schedule function uses epoch argument, epoch not saved, learning rate could get large
                - One solution: set fit() initial_epoch manually to start at right value
            - Alt way to define schedule → use keras.optimizers.schedules
                - Pass learning rate to optimizer
        - Exponential decay, performance scheduling, and 1cycle speed up convergence
- Avoiding Overfitting Through Regularization
    - L1 and L2 regularization

        - Early stopping one of best regularization, batch normalization pretty good too even though not intended as such
        - L2 to constrain weights, L1 to create sparse model
        - Want to apply regularizer to all layers and use same activation function, use functools.partial() to avoid errors
    - Dropout
        - Even state-of-the-art NN get 1-2% accuracy boost from dropout
        - At every training step, every neuron has probability of being dropped out, excludes output neurons but not input
        - Dropout rate typically set between 10-50%, 20-30% for RNN
        - Company analogy: do more with less!
        - Unique NN at each training step, 2^N possible networks, means next to impossible for same NN to be sampled twice
            - 10k training steps → 10k NNs → End NN ensemble of smaller NN
            - In practice usually only dropout on top 1-3 layers
        - Technical detail: if prob = 50%, in testing each neuron connected to 2x as many input neurons as in training
            - Multiply weights by keep probability (1 – prob) after training
            - Keras.layers.Dropout randomly drops some % of inputs and divides remainder by keep probability
        - Since dropout active only during training evaluate training loss w/out dropout
        - Dropout slows convergence
        - Regularize self-normalizing network based on SELU, use alpha dropout → preserves mean stdev of inputs
    - Mote Carlo (MC) dropout
        - Connection between dropout networks and Bayesian inference
        - MC dropout can boost performance of trained dropout model w/out having to retrain or modify
        - Number of MC samples is hyperparameter, higher → more accurate, but inference time doubled too. Improvement slows after some # of samples
    - Max-Norm Regularization
        - Constrains incoming weights for each neuron $||\mathbf{w}||_2 \leq r$, where r is max-norm hyper parameter and $||*||_2$ is l2 norm
        - Doesn't add regularization loss to loss function
            - Computes $||w||_2$ after each training and rescales w
                - $w \leftarrow w\, r/||w||_2$
            - Reduce r, regularization↑, overfitting ↓
- Summary and Practical Guidelines
    - Author recommendations

Default DNN configuration

Hyperparameter	Default value
Kernel Initializer	He Initialization
Activation function	ELU

Normalization	None if shallow: batch norm if deep
Regularization	Early stopping (+ l2 if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

DNN configuration for a self-normalizing net

Hyperparameter	**Default value**
Kernel Initializer	LeCun Initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

- Don't forget to normalize input features
- Exceptions to guidelines
  - Need a sparse model → use l_1 regularization
  - Need a low latency model → fold Batch Normalization layers into previous layers and use faster activation function like leaky ReLU; reduce float precision