Training Deep Neural Networks

The Vanishing/Exploding Gradients Problems

- When training DNNs, gradients often get smaller as the algorithm progresses to the lower layers.
    - The gradient descent algorithm leaves the lower layers' connection weights unchanged, so training never converges to a good solution.
    - This is called the vanishing gradients problem.
- The opposite can occur, too: the gradients can grow bigger until the layers get huge weight updates and the algorithm diverges.
    - This is called the exploding gradients problem.
- The leading cause for these problems was the combination of the weight-initialization technique and the sigmoid function.
    - Before, weights were initialized with a mean of 0 and a standard deviation of 1.
    - The curve begins to flatten as the sigmoid function approaches 0 or 1.
    - These attributes made it so that little to no gradient could propagate through the network; in other words, the signal weakened during training.

Glorot and He Initialization

- For the signal to flow properly, we must have two things:
    - The variance of each layer's outputs must be equal to the variance of its inputs.
    - The gradients must have equal variance before and after flowing through a layer in the reverse direction.
    - It is impossible to guarantee both unless the layer has an equal number of inputs and outputs (the fan-in and fan-out of the layer).
    - Also, the connection weights of each layer must be initialized randomly.
- We can use the Glorot initialization instead of the sigmoid function.
    - $fan_{avg} = (fan_{in} + fan_{out})/2$
    - The function has a normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$
    - Replacing fan_{avg} with fan_{in} yields the LeCun initialization.
        - This is the chosen initialization method for the SELU function.
        - Glorot = LeCun when fan_{in} = fan_{out} .
- We can use the He initialization or Kaiming initialization instead of the ReLU function.
    -

Initialization	Activation Functions	Variance (Normal)
Glorot	None, tanh, sigmoid, softmax	$1/fan_{avg}$

He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2/fan_{in}$
LeCun	SELU	$1/fan_{in}$

- We can change which initialization method Keras uses by specifying the *kernel_initializer* parameter.
  - By default, Keras uses Glorot initialization with a uniform distribution.
  - When we create a layer, we can specify the initialization to “he_normal” to use He initialization.

Better Activation Functions

- Although the ReLU function is a good default, it has some limitations.
  - It suffers from the dying ReLUs problem: some neurons can die during training, meaning they output only zeros.
  - We can use ReLU variants to solve this problem.
- Leaky ReLUs ensure that neurons never die.
  - Function: $LeakyReLU_{\alpha}(z) = max(\alpha z, z)$
  - The hyperparameter α controls how much the function “leaks”: it is the slope for the function z < 0.
    - Having a slope for z < 0 ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up.
  - With a randomized leaky ReLU (RReLU), α is set randomly.
    - RReLU performs well as a regularizer, reducing the risk of overfitting.
  - With a parametric leaky ReLU (PReLU), α can be learned during training.
    - PReLU outperforms ReLU on large datasets but risks overfitting on smaller ones.
  - Keras has **LeakyReLU()** and **PReLU()** classes that we can use instead of ReLU.
    - When building a layer, we simply specify the *activation* parameter to one of these classes.
    - There is no class for RReLU, but we can implement this ourselves.
- ELU and SELU are two other activation functions we can use instead of ReLU.
  - The equation for an exponential linear unit (ELU) is:
    - $ELU_{\alpha}(z) = \alpha(exp(z) - 1)\ if\ z < 0,$

      $z\ if\ z \geq 0.$
    - If α = 1, the function is smooth everywhere, which helps speed up gradient descent by minimizing the amount of bouncing at z = 0.
    - We can use ELU by specifying “elu” as the activation function in Keras.
    - ELU is slower to compute than the ReLU function and its variants.
  - If all the hidden layers use the SELU activation function, the network will self-normalize.

        - The output of each layer will tend to preserve a mean of 0 and a standard deviation of 1 during training.
        - We can use SELU by specifying “selu” as the activation function in Keras.
        - There are a few conditions for self-normalization:
            - The input features must be standardized: mean = 0, std = 1.
            - Every hidden layer’s weights must be initialized using LeCun normal initialization (e.g., set kernel_initializer to “lecun_normal”).
            - Self-normalization only works with plain MLPs.
            - We cannot use regularization techniques, such as ℓ1 or ℓ2, max-norm, batch-norm, or dropout.
- GELU, Swish, and Mish are smooth variants of ReLU.
    - For the GELU activation function:
        - $GELU(z) = z\phi(z)$ where ϕ is the standard Gaussian cumulative distribution function.
        - GELU’s complex shape and presence of curvature at every point makes it easier for gradient descent to fit complex patterns.
        - GELU is more computationally intensive, and the performance boost may not be sufficient to justify the extra cost.
        - We can specify activation=”gelu” to use GELU in Keras.
    - The Swish function introduces a β hyperparameter to scale the sigmoid function’s input.
        - $Swish_{\beta}(z) = z\sigma(\beta z)$ and we can tune the β hyperparameter.
        - We can specify activation=”swish” to use GELU in Keras.
    - Mish is another smooth, nonconvex, and non-monotonic variant of ReLU.
        - $mish(z) = ztanh(softplus(z))$, where $softplus(z) = log(1 + exp(z))$.
        - Currently, Keras does not support the Mish function.
- How to decide which activation function to use?
    - ReLU is a good default for simple tasks since it’s very fast to compute.
    - Swish is a better default for more complex tasks.
    - We can use leaky ReLU or PReLU for complex tasks that care about runtime latency.

Batch Normalization

- Although we can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn’t guarantee that they won’t come back during training.
- The batch normalization (BN) technique zero-centers and normalizes each input, then scales and shifts the results using a scaling vector and a shifting vector.
    - In many cases, if we use a BN layer in a neural network, we don’t need to use a StandardScaler() class or a Normalization() layer.
    - BN acts like a regularizer, reducing the need for other reguarization techniques.
    - However, BN adds complexity to the model and the neural network makes slower predictions due to the extra computations at each layer.

    - To use a BN in Keras, we can add a **BatchNormalization()** layer before any of the hidden layers.
        - Occasionally, we might need to tweak the *momentum* hyperparameter, which typically ranges from 0.9 to 0.999.

Gradient Clipping

- Another technique is to clip the gradients during backpropagation so that they never exceed some threshold.
    - In Keras, each optimizer has a *clipvalue* or *clipnorm* argument, which clips every component of the gradient vector to a value between -1.0 and 1.0.
    - The threshold is a hyperparameter that we can tune.

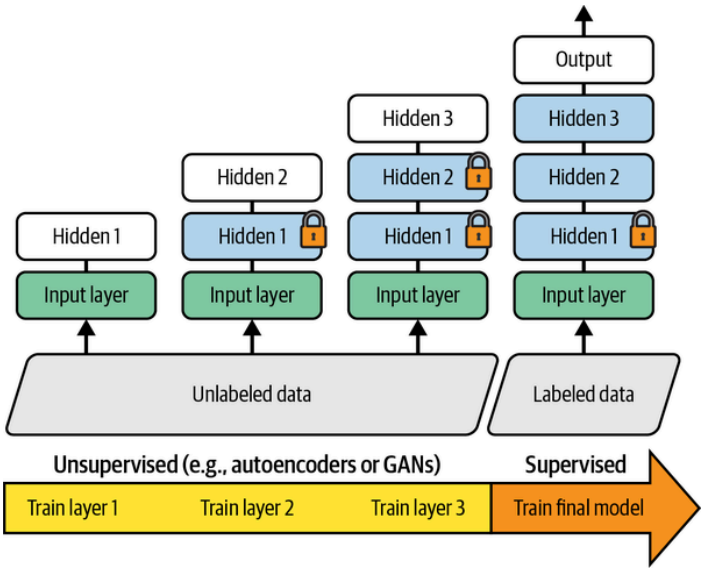
Transfer Learning

- Generally, we don't build a large DNN from scratch; we try to find an existing neural network that accomplishes a similar task.
- If we find such a neural network, we can reuse most of its layers, except its top ones. This is transfer learning.
- To find how many layers would be ideal to reuse, we can try freezing the reused layers first.
    - This means iterating over the model's layers and setting every layer to be untrainable (in Keras, we can set the layer's *trainable* variable to false).
- Then we unfreeze one or two of the top layers to let backpropagation tweak them and see if the performance improves.
    - It may be helpful to reduce the learning rate when unfreezing layers. This will avoid damaging their fine-tuned weights.
- In Keras, we can pass in the existing model's layers as the new model's parameters.
    - Note that this will cause the two models to share layers. So, if we train the new model, it will also affect the first model.
    - To avoid this, we can clone the first model using the **clone_model()** method.
        - Note that this only clones the layers, not the weights. We can get the first model's weights using **get_weights()** and copy them to the second model's weights using the **set_weights()** method.
- It is important to note that transfer learning does not work well with small dense networks.
    - Small networks learn few patterns and dense networks learn very specific patterns, which will unlikely be useful in other tasks.
    - Transfer learning works best with deep convoluted networks, which tend to learn more general feature detectors.

Unsupervised Pretraining

- If we don't have enough labeled data, we can perform unsupervised pretraining.

- It is cheaper to collect unlabeled training data than labeled.
- We can train an unsupervised model with a single layer. Then, we can freeze that layer and add another one on top of it, train the model again, freeze the new layer, add another new layer, and so on. This is called greedy layer-wise pretraining.



Pretraining on an Auxiliary Task

- If we don't have enough labeled training data, we can also try another technique.
  - We train a neural network on an auxiliary task for which we can easily obtain or generate labeled training data.
  - Then, we reuse the lower layers of that network for the actual task.
  - For example, let's say we want to build an facial recognition system.
    - We can combine the few pictures we have and collect pictures of random people on the web.
    - We train a neural network to detect if two different pictures feature the same person.
    - Then, we reuse its lower layers, allowing us to train a good face classifier.

Faster Optimizers

- Momentum speeds up gradient descent.
  - Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the momentum vector m (multiplied by the learning rate η), and it updates the weights by adding this momentum vector.
  - Momentum ranges from 0 (high friction) to 1 (no friction). Momentum is typically set to 0.9.
  - Momentum optimization can reach the optimum much faster than gradient descent.

    - We can set the *momentum* hyperparameter when we build an optimizer in Keras.
- The Nesterov accelerated gradient (NAG) usually helps regular momentum optimization converge faster.
    - We simply set the *nesterov* argument to true when using a Keras optimizer.
- The AdaGrad algorithm helps gradient descent correct its direction as it moves towards the global optimum.
    - It does this by scaling down the gradient vector along the steepest dimensions.
    - This algorithm decays the learning rate fast for steep dimensions. This is called an adaptive learning rate.
    - However, it often stops too early when training neural networks: the learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum
    - Keras has an Adagrad optimizer, but we shouldn't use it for DNNs (though they can be useful for simpler tasks, like linear regression).
- RMSProp fixes AdaGrad by using exponential decay.
    - This allows the algorithm to accumulate only the gradients from the most recent iterations instead of all the gradients since the beginning of training.
    - Keras has an **RMSProp()** optimizer, which uses a learning rate and the decay rate ρ.
- The adaptive moment estimation (Adam) combines the ideas of momentum optimization and RMSProp.
    - It keeps track of an exponentially decaying average of past gradients and exponentially decaying average of past squared gradients.
    - The mean is often called the first moment while the variance is the second moment.
    - Keras has an **Adam()** optimizer, which uses a learning rate, a momentum decay hyperparameter β_1 (often initialized to 0.9), and the scaling decay hyperparameter β_2 (often initialized to 0.999).
    - Adam has three variants: AdaMax, Nadam, and AdamW, all of which Keras supports.
- The table below compares all the optimizers discussed in this chapter (* is bad, ** is average, *** is good).
-

Class	Convergence Speed	Convergence Quality
SGD	*	***
SGD w/ momentum	**	***
SGD w/ NAG	**	***
Adagrad	***	* (stops too early)
RMSProp	***	** or ***

Adam	***	** or ***
AdaMax	***	** or ***
Nadam	***	** or ***
AdamW	***	** or ***

-

Learning Rate Scheduling

- The learning rate controls the speed at which the algorithm converges to the optimum.
- A good way to find a good learning rate is by exponentially increasing the learning rate during training.
- We can also do the opposite: start with a high value and reducing it during training.
- We can use different strategies called learning schedules.
    - Power scheduling: gradually decreasing the learning rate by a factor of s steps.
    - Exponential scheduling: gradually decreasing the learning rate by a factor of 10 every s steps.
    - Piecewise constant scheduling: using a constant learning rate for a number of epochs, then a smaller rate for another number of epochs, and so on.
    - Performance scheduling: measuring the validation error every N steps and reducing the learning rate by a factor of lambda.
    - 1cycle scheduling:
        - We start at an initial rate of η_0, growing linearly to η_1 halfway through training.
        - Then, we decrease the rate linearly down to η_0 during the second half of training.
        - We finish the last few epochs by linearly dropping the rate several orders of magnitude.
- Every optimizer in Keras has a *decay* parameter that we can change.
- We can use the **LearningRateScheduler()** class to create a callback.
- For many of the learning schedules, we would have to implement them ourselves (which is quite simple).

Regularization

- ℓ_1 and ℓ_2 regularization can constrain a neural network's connection weights.
    - We use ℓ_1 regularization with sparse models where many of the weights equal zero.
    - We can access these regularizes with Keras's **regularizers.l1()**, **regularizers.l2()**, or **regularizers.l1_l2()** to use both when building a layer.

    - Note that ℓ_2 regularization works well with SGD, momentum, and NAG, but not with Adam with weight decay. In this case, lets use AdamW.
- Dropout follows a simple algorithm.
    - At every training step, every neuron (including the input neurons but not the output ones) has a probability p of being temporarily "dropped out", meaning it will be entirely ignored during this training step, but it may be active during the next step.
        - p is the dropout rate.
        - In practice, we can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer).
    - With dropout, neurons become less sensitive to slight changes in the inputs, allowing the network to generalize better.
    - Using dropout allows us to train several different neural networks using a single one.
    - For dropout to work, we need to divide the connection weights by the keep probability (1-p) during training.
    - Keras has a **Dropout()** layer, which we can add after every layer in the network.
    - Note that the training loss and validation loss can be misleading. We have to evaluate the training loss after training (without dropout) to accurately judge the network's performance.
        - If the model overfits, we can increase the dropout rate. Likewise, if the model undefits, we can decrease the dropout rate.
- Monte Carlo (MC) Dropout
    - MC dropout can boost the performance of a trained dropout model without retraining it or modifying it.
    - In fact, we can implement MC dropout using only NumPy.
        - We can also make a function that implements MC dropout by passing Keras's Dropout as a parameter.
    - Note that MC dropout tends to improve the reliability of the model's probability estimates, meaning that they are more prone to false positives.
- Max-Norm Regularization
    - For each neuron, max-norm constrains the weights w of the incoming connections such that $\| w \|_2 \leq r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ2 norm.
        - In other words, max-norm applies both ℓ2 regularization and an r hyperparameter.
        - Decreasing r increases the amount of regularization, helping reduce overfitting.
    - We can set Keras's *kernel_constraint* parameter to **keras.constraints.max_norm()** function when building a hidden layer.
        - Keras also has a *bias_constraint* parameter, which we can tweak.

Practical Guidelines

- Below is a table for the default DNN configuration.

-

Hyperparameter	Default Value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; batch norm if deep
Regularization	Early stopping; weight decay if needed
Optimizer	NAG or AdamW
Learning rate schedule	Performance of 1cycle

-

- For a self-normalizing DNN:
  -

Hyperparameter	Default Value
Kernel initializer	LeCunn initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	NAG
Learning rate schedule	Peformance or 1cycle

  -