**Deep Learning with R – Chapter 1**

* The central problem in machine learning is to *meaningfully* transform data: how can we provide the computer with meaningful representations.
* *Learning,* in the context of machine learning, describes an automatic search process for better representations.
* Deep learning is a specific subfield of machine learning that puts an emphasis on learning successive layers of increasingly meaningful representations.
  + The number of layers contributing to a model of the data is called the *depth* of the model.
* Deep learning is, technically, a multistage way to learn data representations.
* What a layer does to its input is determined by its weights, parameters that must be learned through exposure to the data.
  + Learning means finding the set of values of all layers in a network, such that network will correctly map example inputs to their associated targets.
* To control the output of a neural network, you need to be able to measure how far this output is from what you expected. This is the job of the *loss function* of the network.
  + The loss function takes the predictions of the network and the true target and compute a distance score, capturing how well the network has performed.
  + The value of the loss function is then used to adjust the weights of the network in the correct direction.
* Together with performance gains, deep learning also automates what used to be the most crucial step in a machine learning workflow: feature engineering.

**Deep Learning with R – Chapter 2**

* In machine learning, a category in a classification problem is called a *class*. Data points are called *samples*. The class associated with specific samples is called a label.
* Basic machine learning workflow:
  + We feed the neural network the training data.
  + The network will then learn to associate images and labels.
  + We ask the network to produce predictions for the test data and we then verify whether these predictions match the labels from test labels.
* The core building of a neural network is the layer, a data-processing module that you can think of as a filter for the data. Some data goes in, and it comes out in a more useful form.
  + Layers extract *representations* of the data that is fed into them.
* Tensors are a generalization of vectors and matrices to an arbitrary number of dimensions.
  + A scalar is a tensor that contains only one number.
  + A one-dimensional array of numbers is called a vector, or 1D tensor.
  + A matrix is a two-dimensional array of numbers (a 2D tensor). A matrix has two axes (often referred to as *rows* and *columns*).
    - If you pack such matrices in a new array, you obtain a 3D tensor, which you can visually interpret as a cube of numbers.
  + A tensor is defined by three key attributes
    - Number of axes (rank): a 3D tensor has three axes, and a matrix ha two axes.
    - *Shape*: an integer vector that describes how many dimensions the tensor has along each axis.
    - *Type*: The type of data contained in the tensor; for instance, a tensor’s type could be *integer* or *double*.
  + In general, the first axis in all data tensors you’ll come across in deep learning will be the *samples* axis.
  + Deep-learning models don’t process an entire dataset at once; rather, they break the data into small batches.
* Vector data is the most common case of tensors. In such a dataset, each single data point can be encoded as vector. Thus, a batch of data will be encoded as a 2D tensor (an array of vectors), where the first axis is the *sample axis* and the second axis is the *feature axis*.
* Time or sequence data is stored in a 3D tensor:
  + The first axis is the sample axis, and can represent, for example, one trading day.
  + Each sample has more than one place in time. For each minute of a trading day, we could have 390 minutes of price data (this is the time axis).
  + For each sample in a time sequence, we might have up to *m* features representing characteristics of the sample at that particular moment.
* Image data is stored in a 4D tensor:
  + The first axis is the sample axes and represents each individual sample.
  + The next two axes represent height and width of each image.
  + The fourth axis represents the color depth of each sample.
  + All transformations learned by deep neural networks can be reduced to a handful of tensor operations applied to tensors of numeric data.
  + For instance, the **relu** operation is an element-wise operation: operations that are applied independently to each entry in the tensors being considered.
  + Operations involving tensors of different dimensions involve *sweeping* the operation across one of the dimensions of the largest tensor. For instance, to add a vector to a matrix, we could add the vector to all columns of the matrix or add the vector to all the rows of the matrix.
  + The dot operation, also called *tensor* *product* combines entries in the input tensors (a dot product, basically, is the projection of one vector over another). The larger a dot product, the smaller the angle between them (the larger the projection of one over the other).
  + Reshaping is another common tensor operation. Reshaping a tensor means rearranging its rows and columns to match a target shape. A special case of reshaping is transposing a matrix, so its rows become it columns and vice-versa.
  + All tensor operations have geometric interpretations.
  + Remember that applying a matrix to a vector means finding new coordinates for that vector in a space defined by the matrix.
  + Neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just geometric transformations of the input data. It follows that you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a long-series of simple steps.
  + Deep learning provides a way to decompose a very complicated geometric transformation a long chain of elementary ones.
    - Each layer disentangles the data a little – and a deep stack of layers makes tractable an extremely complicated disentanglement process.
  + Each layer of a neural network contains attributes called *weights* or *trainable parameters*.
    - Initially, we assign random values to these parameters. Although the representation resulting from this random transformation will be meaningless, they will allow us to gradually adjust them.
    - We update the weights using the derivative of the loss function with respect to the weights.
      * The gradient of any function is a vector has the direction of the greatest increase of the function at a point *f* (x), while its magnitude is the rate of increase in that direction.
      * By moving in the opposite direction of the gradient, we are, in effect, minimizing *f* (x)
  + The optimal parameters, that is, the weights that minimize the mismatch the loss function could be found, theoretically, analytically. If the loss function is convex, then its minimum would be at the value that its derivative is equal to 0. However, this is intractable for humans after a certain point, because that would mean finding the partial derivatives of thousands of individual weights.
  + Stochastic gradient descent is a method that involves calculating the gradient of the loss function in a random batch of data and updating the weights based on the results.
  + Description gradient descent:
    - Draw a batch of training samples x and corresponding targets y
    - Run x through the network to obtain predictions (y\_pred)
    - Compute the loss of the network on the batch, i.e., a measure of mismatch between y\_pred and y.
    - Compute the gradient of the loss with regard to the network’s parameters (*a backward pass*)
    - The gradient is a vector and contains a value for each weight in W (the direction in that dimension). Add this new vector to your original weights so you “move then” in the opposite direction, therefore minimizing the loss.
    - When updating the weights, analysis usually multiply the gradient by a learning parameter (the *step factor*), which “smoothes” the changes in the weights, preventing them from:
      * Taking too long to converge (if the step is too small)
      * Or never converging.
  + Momentum is a concept utilized in some versions of gradient descent algorithms aimed at preventing the algorithm from getting stuck in a local minimum. Momentum updates the weights of a neural network based not only on the current value of the gradient, but also on the previous parameter update.
  + Since each neural network can be understood as a chain of transformations, we can calculate the derivative of the loss function with respect to each step. Based on the loss function, we first update the weights in the last layer and work backwards until the first.
* An iteration over all the training data is called an *epoch*
* Summary of two key concepts:
  + The loss is the quantity that you’ll attempt to minimize during training, so it should represent a measure of success for the task you are trying to solve.
  + The optimizer specifies the exact way in which the gradient of the loss will be used to update parameters.

**Deep Learning with R – Chapter 3**

* Neural networks workflow:
  + Layers map the input data to predictions
  + The loss function compares these predictions to the *targets*, producing a loss value
  + The optimizer uses this loss value to update the network’s weights.
* A layer is a **data-processing module** that takes as input one or more tensors and that outputs one or more tensors. Different layers are appropriate for different tensor formats.
* Building deep-learning models is done by clipping together compatible layers to perform useful data transformations.
* By choosing a network topology, the analyst effectively constrains what transformations are possible.
* After defining a network topology, it is necessary to define:
  + A loss function: the quantity that will be minimized during training.
  + An optimizer: an implementation of a specific variation of stochastic gradient descent.
* **Classifying movie reviews** **(Use case 1)**:
  + In the IMDB dataset, each review (sequence of words) has been turned into sequences of integers, where each integer stands for a specific word in a dictionary.
  + To put the reviews into a neural network, it is necessary to turn the reviews into tensors. There are two ways of doing this:
    - Pad the lists, so they have the same length, turn them into an integer tensor of shape (samples, word\_indices), and then use as the first layer in your network a layer capable of handling integer tensors.
    - One-hot encode the lists to turn them into vectors of 0s and 1s.
    - The input data is a vector: each review was transformed into a vector. The labels are scalers (1 and 0) representing review type.
    - A type of network that works well with this type of application is a simple stack of fully connected layers with relu activation.
      * *A relu activation means that each neuron will receive its inputs and apply a rectifier, defined as*
        + **.** The output is simple: if x is greater than 0, than the node will output the input value, otherwise it will output 0.
        + A fully connected layer means that every output is a function of every input.
    - In this application each layer will have 16 hidden units. This means that the weight matrix *W* will have 16 components.
      * Adding more hidden units might lead to overfitting data.
    - The question that every modeler face is about the number of hidden units to choose in each layer, and the number of layers itself.
  + In the absence of activation functions, the network would only be able to learn linear transformation of the input vector. By transforming the inputs using non-linear functions, we are able to explore a larger hypothesis space.
  + After deciding on the model, it is necessary to choose the loss function that the algorithm will have to minimize. For this application we will use binary cross-entropy.
    - Cross entropy loss measures the performance of a classification whose output is a probability between 0 and 1.
  + To monitor training accuracy, we will create a validation set by setting apart 10,000 samples from the original training data.
  + The model will be trained for 20 epochs (20 iterations over all samples in x\_train and y\_train tensors), in mini-batches of 512 samples:
    - From the training data, randomly draw 512 samples and update the weights until all training data has been used. Do that 20 times.
  + The loss plot shows that the loss is minimized around the fourth epoch. After that, each update to the weights does not improve accuracy on the validation set.
  + Example wrap-up:
    - Relu is a widely used activation function.
    - In a binary classification problem, the network should end with a dense layer with one unit and a *sigmoid* activation.
      * As a consequence, the loss function should be binary cross-entropy.
* **Classifying newswires (use case 2)**
  + This topic classification problem is similar to sentiment analysis, but there is an additional difficult: the number of output classes increased from 2 to 46.
    - To deal with this problem, we might want to create denser layers.
    - Small layers in complicated problems might function as information bottlenecks.
  + The activation function in the output layer is a *softmax* function. This function takes a vector of of length K, normalizes it into a probability distribution where each dimension corresponds to the probability of the sample belonging to that class.
  + The loss function that we are optimizing over is *categorical cross-entropy*. It measures the distance between two probability distributions: the vector outputted by the network and the true distribution of the labels.
  + Using layers that are significantly less dense might result in a loss of accuracy due to the fact that you will be trying to compress into an intermediate space that is too low-dimensional.
  + Extra: training a model with 128-hidden units:
    - The model’s accuracy peaked at the last epoch. However, since the fourth or fifth epoch, the accuracy/loss of the model stabilized. The improvement over the simpler model (64-hidden units was negligible).
* **Predicting housing prices (use case 3)**
  + Another common type of machine learning problem is *regression,* which consists of predicting a continuous value instead of a discrete label.
  + The dataset used in the example consists of the median price of homes in a Boston suburb in the mid-1970s.
  + Given that each feature is on a different scale, it is good practice to normalize it, so they have standard deviation of 1 and mean of 0. The values used for normalization should always come from the training data.
  + The network has no activation function in the output layer, because we do not want to constrain the range of the output.
  + The loss function is *mean squared error* (the square of the difference between the predictions and the target).
  + Given that the dataset is so small, leaving aside a single validation set to track the model would generate highly variable estimates. The way to mitigate this problem is using K-fold cross-validation:
    - K-fold cross-validation consists of splitting the available data into K partitions, instantiating K identical models, and training each one on K-1 partitions while evaluating on the remaining partition.
  + The model calculates the weights on 303 samples and uses that to predict the left-out sample. After the model goes through each sample 100 times (number of epochs), the trained weights will be used to predict the values in the validation set. This is done 4 times. Finally, the model is averaged over.
  + The model stops improving after 39 epochs. So, we then try a simpler model with 39 epochs.
  + When little training data is available, it is preferable to use a small network with few hidden layers in order to avoid severe overfitting.
* If your data is divided into many categories, you may cause information bottlenecks if you make the intermediate layers too small.

**Deep Learning with R – Chapter 4**

* Machine learning generally fall into one of four broad categories:
  + 1) Supervised learning: consists of learning to map input data to known targets given a set of examples.
  + 2) Unsupervised learning: consists of finding interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data denoising, or to better understand the correlations present in the data at hand.
  + 3) Self-supervised learning: similar to supervised learning but with target labels generated from the input data, typically using a heuristic algorithm.
  + 4) Reinforcement learning: an *agent* receives information about its environment and learns to choose actions that will maximize some reward.
* A batch is a small set of samples that are processed simultaneously by the model. The number of samples is often a power of *2*, to facilitate memory allocation on GPU.
* The goal of machine learning is to achieve models that generalize – that perform well on never-before-seen data. Therefore, overfitting must be avoided.
* Every time you tune a hyperparameter of your model based on the model’s performance on the validation set, some information about the validation data leaks into the model. If you keep optimizing the model to achieve a good performance on the validation data, you might end up tuning the model to the validation dataset.
  + What is important is never before seen data, that is why we also utilize a test dataset to evaluate performance.
* **Simple hold-out validation**:
  + Set apart some fraction of the data as a test set. Train on the remaining data and evaluate on the test set. To prevent information leaks, model should be tuned based on a validation set.
  + Usually requires a large amount of data so the validation set is representative of the data at hand.
* **K-Fold Validation**
  + Split the data into K partitions of equal size. For each partition *i,* train a model on the remaining K – 1 partition, and evaluate on the partition *i*. The model performance is then an average of the K scores obtained.
  + This method is helpful when the model performance shows significant variation based on your training/test split.
* **Iterated K-Fold Validation with Shuffling**
  + Consists of applying K-fold validation multiple times, shuffling the data every time before splitting it K ways. The final score is an average of the scores obtained at each run of K-fold validation.
* Data preprocessing aims at making the raw data at hand more amenable to neural networks.
  + **Vectorization**
    - All inputs and targets in a neural network must be tensors of floating-point data. Whatever data that is needed to process must first by turned into tensors.
  + **Value Normalization**
    - To make learning easier for the network and preventing large gradient updates that will make it harder for the model to converge, your data should have the following characteristics:
      * Take small values: usually in the 0-1 range
      * Be homogenous: all features should roughly be in the same range
    - Typically, feature normalization happens in both the training and in the test datasets. Compute the mean and std. deviation on the training data and apply both values to the training and test data.
* Feature engineering is the process of using knowledge about the data and the algorithm to optimize its behavior by applying hardcoded transformations to the data before it goes into the model.
* The fundamental issue in machine learning is *generalization* and *optimization.*
* After a certain number of iterations, the model is learning patterns that are specific to the training data (it starts to overfit).
* There are two ways of dealing with overfitting:
  + Getting more training data.
  + Regularization: modulating the quantity of information that the model is allowed to store or add constraints on what information it’s allowed to store.
* The simplest way to fight overfitting is reducing the size of the model: the model of learnable parameters in the model. In a neural network, that means adjusting the number of layers and the number of hidden units in each layer.
* On the other hand, the network must have enough parameters so it can learn valuable patterns.
* The best procedure is to start with relatively few layers and parameters and increase the size of the layers or add new layers until you see diminishing returns.
* A *simple* model is a model where the distribution of parameter values has less entropy.
  + A common way to mitigate overfitting is by forcing the weights to take only small values, which makes the distribution of weights more regular. This is done by adding to the loss function of the network a cost associated with having large weights.
* *Dropout* is another technique for regularization that consists of randomly setting to zero a number of output features of the layer during training. Dropout is controlled by a dropout rate that refers to the fraction of the features that are zeroed out.
  + *Dropout* refers to ignoring neurons during the training phase of certain set of neurons performed at random. During training, for each hidden layer, for each training sample, ignore a random fraction of nodes. During the testing phase, reduce them by a factor of *p.*
* Defining a last layer activation is necessary to constrain the network’s output to a certain range.

**Deep Learning with R – Chapter 5**

* Remember that image data is stored as a 4D tensor. The first index represents the sample; the second represents the height; the third represents the width; the fourth represents the color depth of each image.
* The fundamental difference between a densely connected layer and a convolution layer is that dense layers learn global patterns in their input feature space, whereas convolution layers learn local patterns. In the case of images, patterns found in small 2D windows of the inputs.
* The patterns learned from convnets are translation invariant: after learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere.
  + Moreover: convnets can learn *spatial hierarchies of patterns*. A first layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of features of the first layer, and so on.
* Convolutions operate over 3D tensors, with two spatial axes (*height* and *width*) as well as a *depth* axis with length corresponding to the number of color channels (for a B-and-W image, the depth is 1 (levels of gray)); for an RGB image, the depth is 3, because there are 3 color channels (one for RED, other for GREEN, and other for BLUE).
* The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an output feature map.
  + The output feature map still contains width and height, but the depth is a hyperparameter defined by the modeler.
* Convolutional neural networks are mapping an input tensor to an output feature map by “scanning” the input and computing features at each section of the input.
* Output and input feature maps might differ in their dimensions. To ensure that they are of the same size, it is possible to use *padding*.
  + *Padding* consists of adding an appropriate number of rows and columns on each side of the input feature map to make it possible to fit center convolution windows around every input tile.
* *Max-pooling* layers perform *max-pooling* operations, meaning that they aggressively downsample feature maps. Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel.
* The reason for downsampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows.
  + The first layer looks at certain windows and outputs something. The output map shrinks the output tensor and sends it to another convolutional layer that looks at another area of the shrunk tensor.
* Strategies for computer vision problems with small datasets:
  + 1) Training a small model from scratch
  + 2) Feature extraction using a pretrained model
  + 3) Fine-tuning a pretrained model.
* Deep learning using convnets can still be useful because the networks learn local, translation-invariant features.
* Training a convnet from scratch on a small dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering.
  + Remember that, roughly speaking, deep learning involves the networks transforming the input tensors so it can learn meaningful representations.
* Data needs to be preprocessed into floating point tensors before being fed into the network. The workflow then consists of:
  + 1) Read the picture files
  + 2) Decode the JPEG Content to RGB grids of pixels
  + Convert these into floating-point tensors
  + Rescale the pixel values to the [0-1] interval.
* In KERAS, you are defining the batch size in the generator.
* In image classification, a very common regularization technique is called *data* augmentation.
* Data augmentation takes the approach of generating more training data from existing samples, by *augmenting* the samples via a number of random transformations that yield believable looking images. The goal is that at training time, the model will never see the exact same picture twice.
* A common and highly effective approach to deep learning on small image datasets is to use a pretrained network.
  + *A pretrained network* is a saved network that was trained on a large dataset. If this dataset is large enough and general enough, then the spatial hierarchy of features learned by the pretrained network can effectively act as a generic model of the visual world.
* Two ways of using a pretrained convolutional neural network:
  + 1st) Feature extraction consists of using the representations learned by a previous network to extract interesting features from the new samples. These features are then run through a new classifier, which is trained from scratch (basically use the convolutional part of the architecture).
    - The feature maps learned by convnets are presence maps of generic concepts over a picture.
    - It is important to keep in mind that the representations extracted by specific convolution layers depend on the depth of the layer. So, if the new dataset is very different from the original training dataset, you will be better off using only the first few layers of the model to do feature extraction.
* Freezing a layer or set of layers means preventing their weights from being updated during training.
* Another widely used technique for model reuse is *fine-tuning*. Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model and the unfrozen layers.
* Convnets wrap-up:
  + 1st) Convnets are the best type of machine-learning models for computer-vision tasks. It is possible to train one from scratch even on a very small dataset, with decent results.
  + 2nd) On a small dataset, overfitting is the main issue. Data augmentation is a powerful way to fight overfitting when working with image data.
  + 3rd) It is possible to reuse other models for feature extraction and build classifiers on top of it.
* The representations learned by convnets are highly amenable to visualization, in part because they are representations of visual concepts.
* **Visualizing intermediate activations**
  + Visualizing intermediate activations consists of displaying the feature maps that are output by various convolution and pooling layers in a network given a certain input.
    - This gives a view into how an input is decomposed into the different filters learned by the network.
    - The features extracted by a layer become increasingly abstract with the depth of the layer. The activations of higher layers carry less and less information about the specific input being seen, and more information about the target.
* **Visualizing convnet filters**
  + Consists of displaying the visual patterns that each filter is mean to respond to.
    - This is done using *gradient ascent* *in input space*: applying gradient descent to the value of the input image to maximize the response of a specific filter, starting from a blank input image.
      * The process consists of building a loss function that maximizes the value of a given filter in a given convolution layer, and then you’ll use stochastic gradient descent to adjust the values of the input image so as to maximize this activation value.

**Deep Learning with R – Chapter 6**

* The two fundamental deep-learning algorithms for sequence processing are *recurrent neural networks* and *1D convnets*.
* Deep learning for natural-language processing is pattern recognition applied to words, sentences, and paragraphs.
* Vectorizing text so it can be understood by neural networks can be done in multiple ways:
  + 1st) Segment text into words and transform each word in a vector.
  + 2nd) Segment text into characters, and transform each character in a vector.
  + 3rd) Extract n-grams of words or characters, and transform each n-gram into a vector.
* Word n-grams are groups of N (or fewer) consecutive words that you can extract from a sentence.
* The term bag-of words refers to the fact that we are dealing with a set of tokens rather than a list or sequence.
* One-hot encoding is the most common way to turn a token into a vector. It consists of associating a unique integer index with every word and then turning this integer index *i* into a binary vector of size *N*. The vector is all zeros except for the *i*th entry, which is 1.
  + Roughly speaking, each document is a matrix of 1s and 0s based on the vocabulary used.
* Another popular to associate a vector with a word is the use of dense word vectors, also called *word embeddings*.
  + Word embeddings are learned from data and are low-dimensional floating-point vectors.
  + There are two ways to learn word embeddings:
    - Learn word embeddings jointly with the main task you care about (such as document classification). In this task, you start with random word vectors and then learn word vectors in the same way you learn weights of a neural network.
    - Load into your model word embeddings that were precomputed using a different machine-learning task than the one you are trying to solve.
  + Word embeddings should, in a way, meaningfully represent relationship between words in a geometric space. Synonyms should be embedded into similar word vectors.
    - In addition to distance, you may want specific *directions* in the embedding space to be meaningful.
      * Applying similar matrices to different vectors should lead to similar results. For instance, the same transformation that leads from “king” to “man” should lead “female” to “woman”.
  + A good word-embedding representation depends heavily on the task at hand. Therefore, it is reasonable to learn a new embedding space with every new task.
  + Learning word-embeddings requires instantiating an embedding layer, which takes as input a sequence of integers and returns a corresponding word vector.
    - Basically, it returns a position into the “language” space that has the same dimensionality as the output\_dim parameter.
    - Each sequence is transformed into a sequence of vectors.
* Instead of learning word embeddings jointly with the problem you want to solve, it is possible to load embedding vectors from a precomputed embedding space that you know is highly structured and exhibits useful properties.
* *Feed-forward* networks process each input independently. To process a sequence or a temporal series at once, it is necessary to convert them into a single data point.
  + Nevertheless, biological intelligence processes information incrementally while maintaining an internal model of what it’s processing, built from past information and constantly updating as new information arrives.
* A recurrent neural network tries to imitate this process by iterating through the sequence elements and maintaining a *state* containing information relative to what it has seen so far.
* Intuitively, we consider RNNs as a for loop that reuses quantities computer during the previous iteration of the loop. Each data point is not processed in a single step anymore (the network loops over the sequence elements).
  + The state of the network is reset when a new data point starts to be processed.
* Each sequence is processed independently, and we then save the output of the network in a vector of *states*.
* If we consider, for example, a one-hot encoding, we would multiply each word individually and save the results obtained.
* Keras implements RNNs with the command layer\_simple\_rnn:
  + It can return the successive outputs for each timestep (a 3D tensor of shape (batch\_size, timesteps, output\_features)) or only the last output for each input sequence (a 2D tensor of shape (batch\_size, output\_features)).
* It is sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of the network. In such a setup, it is necessary to set all intermediate layers to return full sequences.
* layer\_simple\_rnn has a shortcoming associated with the *vanishing gradient problem*. Therefore, it has problems retaining information learned early in the process.
  + To circumvent this problem, the LSTM layer is a variant of the RNNs that saves information for later processing, thus preventing older signals form gradually vanishing during processing.
  + The LSTM layer allows past information to be reinjected at a later time, thus fighting the vanishing-gradient problem.
* A generator function is a special type of function that you call repeatedly to obtain a sequence of values.
* A significant limitation of machine learning is that unless the learning algorithm is hardcoded to look for a specific kind of simple model to improve upon, parameter learning can sometimes fail to find a simple solution to a simple problem.
* GRU layers are similar to LSTM, but they are streamlined and cheaper to run.
* The application of dropout to recurring neural networks requires that the same dropout pattern be applied to every step, instead of a dropout pattern that varies randomly from step to step.
  + Every recurrent layer in Keras has two dropout-related arguments: dropout, a float specifying the dropout rate for input units of the layer, and recurrent dropout specifying the dropout rate of the recurrent units.
* RNNs are notably order dependent, or time dependent: they process the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the representations the RNNs extracts from the sequence.
* A bidirectional RNN exploits the order sensitivity of RNNs: it consists of using two regular RNNs, each of which processes the input sequence in one direction, and then merging their representation.
  + This should work well in NLP tasks, because the importance of a word in understanding a sentence is not usually dependent on its position in the sentence.
  + On the same IMDB dataset task, the reversed-order processing performed as well as the regular implementation, providing evidence that, although word order does matter, which order is not crucial.
* In machine learning, representations that are different yet useful are always worth exploring, and the more they differ, the better: they offer a new angle from which to look at the data, capturing aspects that were missed by other approaches.
  + A bidirectional RNN exploits this idea to improve the performance of chronological order RNNs. It looks at its input sequences both ways, obtaining potentially richer representations and capturing patterns that may have been missed by the chronological order version alone.
* Sequence processing can also be performed with convolutional neural networks.
* For image processing, convolutional networks extracted 2D patches from image tensors and applying an identical transformation to every patch.
  + Such 1D layers can recognize local patterns in a sequence. Because the same input transformation is performed on every patch, a pattern learned at a certain position in a sentence can later be recognized at a different position, making 1D convnets translation invariant.
* 1D convnets are structured in the same way as their 2D counterparts: they consist of a stack of convolutional and pooling layers, ending in either a global pooling layer or a layer that flats the output of convolution and send them to one or more dense layers to the model for classification or regression.
* In the temperature example, the convnet did not perform as well as the RNN because the layer has no knowledge of the temporal importance of the series. Therefore, it might extract features from the data that are not relevant given that the order of the data points matters in this case.
* One strategy to combine the speed and lightness of convnets with the order sensitivity of RNNs is to use a 1D convnet as a preprocessing step before a RNN.
  + The convnet will make it easier to process the data, since it will turn local input sequences into downsampled windows of data points.
    - Because RNNs are extremely expensive for processing very long sequences, but 1D convnets are cheap, it can be a good idea to use a 1D convnet as preprocessing step before an RNN, shortening the sequence and extracting useful representations for the RNN to process.
    - If global order matters in your sequence data, then it is preferable to use a recurrent network to process it. This is typically the case for timeseries, where the recent past is likely to be more informative than the distant past.
    - If global ordering isn’t fundamentally meaningful, then 1D convnets will turn out to work at least as well and are cheaper. This is often the case for text data, where a keyword found at the beginning of a sentence is just as meaningful as a keyword found at the end.

**Deep Learning with R – Chapter 8 – Generative Deep Learning (Language example)**

* Machine-learning models can learn the statistical *latent space* of images, music, and stories, and then can *sample* from this space, creating new artworks with characteristics similar to those the model has seen in its training data.
* The universal way to generate sequence data in deep learning is to train a network to predict the next token or next few tokens in a sequence, using the previous tokens as input.
  + Any network that can model the probability of the next token given the previous ones is called a language model.
  + After training a language model, you can feed it an initial string of text, and ask it to generate the next character or the next word. This is possible because for each input, the model has a probability distribution over all possible tokens in the vocabulary.
  + There are two ways to choose the next token or character in a language model:
    - *Greedy sampling*: always chooses the token or character that has the largest probability of being the next one.
    - *Stochastic sampling*: In this setup, you sample characters according to the probability distribution (Example: if “rat” has a probability of 30%, then in the long run, we expect the language model to choose it 30% of the time).
  + However, it is interesting to control the amount of randomness in the sampling process, because less randomness will generate sequences of more predictable structure, whereas more randomness will introduce more “creative” or unexpected sentences.
    - We are basically adjusting the entropy of the probability distribution generated by the model.
    - In order to control the amount of stochasticity in the sampling process, the model introduces a parameter called the *softmax* temperature that characterizes the entropy of the probability distribution used for sampling it.
      * Given a **temperature** value, a new probability distribution is computed from the original one, reweighting it.
  + Steps to training a language model and sampling from it:
    - 1st) Train a model and generate some text that will serve as a model “query”.
    - 2nd) Draw from the model a probability distribution for the next character, given the generated text available so far.
    - 3rd) Reweight the distribution to a certain temperature.
    - 4th) Sample the next character at random according to the reweighted distribution.
    - 5th) Add the new character at the end of the available text.