

Deep Learning

Onkar Pandit

23 June 2022

Last updated on October 22, 2022.

Contents

1	Introduction	2
2	Basics	3
2.1	Training of neural network	4
2.2	Gradient descent algorithms	5
2.3	Activation Functions	6
2.4	Universal Approximation Properties and Depth	7
2.5	General Concepts	8
2.5.1	The No Free Lunch Theorem	9
2.5.2	Trading off Bias and Variance	9
3	Different Neural Networks	10
3.1	Feedforward networks	11
3.2	Convolutional neural network (CNN)	11
3.3	Recurrent neural network (RNN)	13
3.3.1	LSTM	20
3.4	Attention	23
3.5	Transformers	25
3.6	Autoencoders	32
3.6.1	Denoising Autoencoders	33
3.7	Variational Autoencoders	34
3.7.1	Latent Variable Models	34
3.7.2	Mathematical Intuition	36
3.8	Generative Adversarial Network	36
3.8.1	Architecture	36
3.8.2	Training of Discriminator and Generator	36
4	Regularization	37
4.1	Parameter Norm Penalties	38
4.1.1	L^2 Parameter Regularization	39
4.1.2	L^1 Parameter Regularization	39
4.2	Dataset Augmentation	41
4.3	Early Stopping	42
4.4	Dropout	42

1 Introduction

The performance of these simple machine learning algorithms depends heavily on the representation of the data they are given. For example, when logistic regression is used to recommend cesarean delivery, the AI system does not examine the patient directly. Instead, the doctor tells the system several pieces of relevant information, such as the presence or absence of a uterine scar. Each piece of information included in the representation of the patient is known as a feature.

Logistic regression learns how each of these features of the patient correlates with various outcomes. However, it cannot influence the way that the features are defined in any way. If logistic regression was given an MRI scan of the patient, rather than the doctor's formalized report, it would not be able to make useful predictions. Individual pixels in an MRI scan have negligible correlation with any complications that might occur during delivery.

This dependence on representations is a general phenomenon that appears throughout computer science and even daily life. In computer science, operations such as searching a collection of data can proceed exponentially faster if the collection is structured and indexed intelligently. People can easily perform arithmetic on Arabic numerals, but find arithmetic on Roman numerals much more time-consuming. It is not surprising that the choice of representation has an enormous effect on the performance of machine learning algorithms.

However, for many tasks, it is difficult to know what features should be extracted. For example, suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might like to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the foreground obscuring part of the wheel, and so on.

When designing features or algorithms for learning features, our goal is usually to separate the factors of variation that explain the observed data. In this context, we use the word “factors” simply to refer to separate sources of influence; the factors are usually not combined by multiplication. Such factors are often not quantities that are directly observed. Instead, they may exist either as unobserved objects or unobserved forces in the physical world that affect observable quantities. They may also exist as constructs in the human mind that provide useful simplifying explanations or inferred causes of the observed data. They can be thought of as concepts or abstractions that help us make sense of the rich variability in the data.

When analyzing a speech recording, the factors of variation include the speaker's age, their sex, their accent and the words that they are speaking. When analyzing an image of a car, the factors of variation include the position of the car, its color, and the angle and brightness of the sun.

A major source of difficulty in many real-world artificial intelligence applications is that many of the factors of variation influence every single piece of data

we are able to observe. The individual pixels in an image of a red car might be very close to black at night. The shape of the car’s silhouette depends on the viewing angle.

Most applications require us to disentangle the factors of variation and discard the ones that we do not care about. Of course, it can be very difficult to extract such high-level, abstract features from raw data. Many of these factors of variation, such as a speaker’s accent, can be identified only using sophisticated, nearly human-level understanding of the data. When it is nearly as difficult to obtain a representation as to solve the original problem, representation learning does not, at first glance, seem to help us.

Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. Deep learning allows the computer to build complex concepts out of simpler concepts. For instance, a deep learning system can represent the concept of an image of a person by combining simpler concepts, such as corners and contours, which are in turn defined in terms of edges.

2 Basics

Artificial neural networks are computational systems inspired from the biological neural networks, and *neurons* are the fundamental computation unit in them. However, modern neural network research is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

A neuron processes inputs by applying pre-designed function (generally, non-linear function known as *activation functions*) to produce output. Further, a number of neurons are aggregated into a layer where the first layer of the network always receives input and the final layer produces output, and there can be any number of layers in between which are called *hidden layers*. The neurons, in turn, layers, are interconnected to produce a network or graph, where neurons are nodes and edges denote a connection between them, hence the name *neural network*. The connections are directed labeled edges that indicate the direction and weight of the signal. These weights control the connection and are adjusted at the training step of the network.

A simple neural network is shown in Fig. 1, which contains three layers – *input, hidden, and output layer*, respectively having four, two, and one neurons. A neuron produces output based on the inputs, the associated weights, biases and the *activation function* (Fig. 1 (b)), where weights and biases are called *parameters* of the network and are learned at the training step. A neural network can have any number of layers, any number of neurons in them, and different kinds of connections between them. These are *hyperparameters*¹ which depend

¹Learning rate, learning algorithm, epoch (number of iteration of training), activation

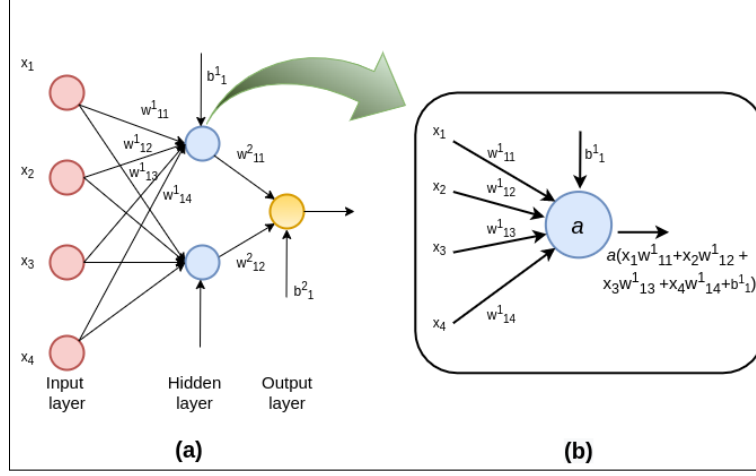


Figure 1: Simple neural network. Fig.(a) shows a simple neural network with three layers - input layer, hidden layer, and output layer where information flows from input layer to hidden layer and finally to output layer. The nodes denote neurons, and edges show a weighted connection between them. A single neuron is detailed in Fig. (b). It takes input from the previous layer and calculates weighted sum which is passed through *activation* function a to produce output.

on the type of problem the neural network solves. In any neural network, the produced output depends on the input and the various transformations applied to it at each layer. Hence, the output \hat{y} obtained from a neural network as a (non-linear) transformation on input x , controlled by parameters θ is given as:

$$\hat{y} = f_{\theta}(x) \quad (1)$$

where f_{θ} is composition of different (non-linear) transformations at various layers: $f_{\theta} = f_1 \circ f_2 \circ \dots \circ f_l$.

2.1 Training of neural network

determines the values of *parameters*. At the beginning of this step, all the parameters, i.e. weights and biases, are randomly initialized. Then the training adjusts these parameters values so as to produce the desired output. The difference between the desired and actual output is measured with a loss function. Suppose, $\mathcal{D} = \{(x_i, y_i)_{i=1}^n\}$ is training data where x_i denote input features and y_i shows associated label. The cumulative loss is given as:

$$\mathcal{L}(\theta) = \sum_{(x_i, y_i) \in \mathcal{D}} l(y_i, \hat{y}_i) \quad (2)$$

function, dropout are some other *hyperparameters*.

where l is a error function which gives difference between true output y_i and predicted output $\hat{y}_i := f_\theta(x_i)$. Then, following optimization objective is solved to get appropriate parameters:

$$\min_{\theta} \mathcal{L}(\theta) \quad (3)$$

The objective function $\mathcal{L}(\theta)$ is non-convex function because of non-linear activation functions and obtaining analytical solution is difficult.

2.2 Gradient descent algorithms

The largest difference between the linear models (SVM, Logistic Regression) and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters (in theory—in practice it is very robust but can encounter numerical problems). Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values.

The gradient descent algorithms are commonly used for minimizing this objective. The algorithm updates parameters of the network iteratively and the parameter values are reduced by the value proportional to the gradient of the loss function with respect to the parameter. The update rule at $(t + 1)^{th}$ iteration is given as:

$$\theta_{t+1} \leftarrow \theta_t - \gamma \frac{\partial \mathcal{L}(\theta)}{\partial \theta} \quad (4)$$

where θ_t denotes the parameters of the neural network at iteration t in gradient descent, and γ learning rate.

However, obtaining a gradient with respect to each parameter in the network is a computationally expensive operation. *Backpropagation* (backward propagation of errors) algorithm is used to calculate the gradient of the error function with respect to the neural network's weights. The gradient calculation proceeds in the backward direction, i.e. first the gradient of the error is calculated with the final layer, then, these values are propagated with the chain rule to obtain gradients with parameters from the previous layers. This produces efficient computation of the gradient at each layer versus the naive approach of calculating the gradient of each layer separately.

Additionally, to speed up the learning process different variants of gradient descent algorithms such as batch, mini-batch, and stochastic gradient descent are used. Recently, various improvements have also been proposed for the gradient descent algorithms: Adagrad, Adadelata, RMSProp, Adam, etc.

2.3 Activation Functions

The input of the neuron is passed through a non-linear function which is known as activation function. Specifically, most neurons can be described as accepting a vector of inputs x , computing an affine transformation $z = W^T x + b$, and then applying an element-wise nonlinear function $g(z)$. Most hidden units are distinguished from each other only by the choice of the form of the activation function $g(z)$.

Most commonly rectified linear function $g(z) = \max\{0, z\}$ is used. There are a family of different activation functions and active research to choose the most suitable activation function. However, it can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice). It is usually impossible to predict in advance which will work best. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

Some of the activation functions are not actually differentiable at all input points. For example, the rectified linear function $g(z) = \max\{0, z\}$ is not differentiable at $z = 0$. This may seem like it invalidates g for use with a gradient-based learning algorithm.

In practice, gradient descent still performs well enough for these models to be used for machine learning tasks. This is in part because neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly. Because we do not expect training to actually reach a point where the gradient is 0, it is acceptable for the minima of the cost function to correspond to points with undefined gradient.

Activations that are not differentiable are usually non-differentiable at only a small number of points. In general, a function $g(z)$ has a left derivative defined by the slope of the function immediately to the left of z and a right derivative defined by the slope of the function immediately to the right of z . A function is differentiable at z only if both the left derivative and the right derivative are defined and equal to each other. The functions used in the context of neural networks usually have defined left derivatives and defined right derivatives.

In the case of $g(z) = \max\{0, z\}$, the left derivative at $z = 0$ is 0 and the right derivative is 1. Software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is undefined or raising an error. This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway. When a function is asked to evaluate $g(0)$, it is very unlikely that the underlying value truly was 0. Instead, it was likely to be some small value ϵ that was rounded to 0. In some contexts, more theoretically pleasing justifications are available, but these usually do not apply to neural network training. The important point is that in practice one can safely disregard the non-differentiability of the hidden unit activation functions described below.

2.4 Universal Approximation Properties and Depth

A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in convex optimization problems when applied to linear models.

Unfortunately, we often want to learn nonlinear functions. At first glance, we might presume that learning a nonlinear function requires designing a specialized model family for the kind of nonlinearity we want to learn. Fortunately, feedforward networks with hidden layers provide a universal approximation framework. Specifically, the universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well. In other words any continuous function on a closed and bounded subset of R^n is Borel measurable and therefore may be approximated by a neural network.

A neural network may also approximate any function mapping from any finite dimensional discrete space to another. While the original theorems were first stated in terms of units with activation functions that saturate both for very negative and for very positive arguments, universal approximation theorems have also been proved for a wider class of activation functions, which includes the now commonly used rectified linear unit. The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function. However, we are not guaranteed that the training algorithm will be able to learn that function.

Even if the MLP is able to represent the function, learning can fail for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function due to overfitting. The “no free lunch” theorem shows that there is no universally superior machine learning algorithm. Feedforward networks provide a universal system for representing functions, in the sense that, given a function, there exists a feedforward network that approximates the function. There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error. There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value d , but which require a much larger model if depth is restricted to be less than or equal

to d . In many cases, the number of hidden units required by the shallow model is exponential in n . Such results were first proved for models that do not resemble the continuous, differentiable neural networks used for machine learning, but have since been extended to these models.

We may also want to choose a deep model for statistical reasons. Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn. Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation. Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output. These intermediate outputs are not necessarily factors of variation, but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better generalization for a wide variety of tasks.

2.5 General Concepts

The central challenge in machine learning is that we must perform well on new, previously unseen inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization. Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the training error, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the generalization error, also called the test error, to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

The train and test data are generated by a probability distribution over datasets called the data generating process. We typically make a set of assumptions known collectively as the i.i.d. assumptions. These assumptions are that the examples in each dataset are independent from each other, and that the train set and test set are identically distributed, drawn from the same probability distribution as each other. This assumption allows us to describe the data generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every test example. We call that shared underlying distribution the data generating distribution, denoted p data. This probabilistic framework and the i.i.d. assumptions allow us to mathematically study the relationship between training error and test error.

The factors determining how well a machine learning algorithm will perform

are its ability to: 1. Make the training error small. 2. Make the gap between training and test error small. These two factors correspond to the two central challenges in machine learning: underfitting and overfitting . Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large. We can control whether a model is more likely to overfit or underfit by altering its capacity. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

One way to control the capacity of a learning algorithm is by choosing its hypothesis space, the set of functions that the learning algorithm is allowed to select as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. Doing so increases the model's capacity.

2.5.1 The No Free Lunch Theorem

for machine learning states that, averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. In other words, in some sense, no machine learning algorithm is universally any better than any other. The most sophisticated algorithm we can conceive of has the same average performance (over all possible tasks) as merely predicting that every point belongs to the same class.

Fortunately, these results hold only when we average over all possible data generating distributions. If we make assumptions about the kinds of probability distributions we encounter in real-world applications, then we can design learning algorithms that perform well on these distributions.

This means that the goal of machine learning research is not to seek a universal learning algorithm or the absolute best learning algorithm. Instead, our goal is to understand what kinds of distributions are relevant to the “real world” that an AI agent experiences, and what kinds of machine learning algorithms perform well on data drawn from the kinds of data generating distributions we care about.

2.5.2 Trading off Bias and Variance

Bias and variance measure two different sources of error in an estimator. Bias measures the expected deviation from the true value of the function or parameter. Variance on the other hand, provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause.

Bias is the difference between the expected prediction of our model and the correct value which is to be predicted. Model has high bias which means that the difference between the correct prediction and expected prediction is very

large, which means that that model rarely gives the right prediction. This is true when the model underfits because it will have high training error as well as test errors.

Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. It is the squared expected distance between the prediction of the model and the expected prediction of the model. If the variance is high, this means that model will vary in the given output a lot. This means, even if we know that the model will give the correct results, because of variability it will produce wrong predictions. Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn't seen before. As a result, such models perform very well on training data but has high error rates on test data.

Let the variable we are trying to predict as Y and other covariates as X . We assume there is a relationship between the two such that

$$y = f(x) + \epsilon$$

Now the expected squared error is calculated as

$$Err(x) = E[(y - \hat{f}(x))^2]$$

which can be written as

$$Err(x) = [E(\hat{f}(x)) - f(x)]^2 + E[(\hat{f}(x) - E(\hat{f}(x)))^2] + \sigma_\epsilon^2$$

In terms of bias and variance it can be written as

$$Err(x) = \text{bias}^2 + \text{variance} + \text{Irreducible Error}$$

Thus we want to achieve low bias as well as low variance. The low bias indicates that the expected predicted value will be close to the true prediction. On the contrary high bias indicates relatively low correct predictions, this means the model underfits the data and makes error most of the time. Thus, we need a model which can produce correct results, therefore we have to increase the capacity of the model, if we increase the capacity too much there is a chance of high variance, this means the expected prediction is itself false from the predicted value. Thus, we have to optimize the complexity of the model so that there is a low bias and low variance.

If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it's going to have high variance and low bias. So we need to find the right balance without overfitting and underfitting the data. This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time.

3 Different Neural Networks

Different types of neural networks produce different transformations of the input data. We describe few popular neural networks that are used in this work as

well.

3.1 Feedforward networks

In these networks, the information flows from the input neurons to hidden neurons to output neurons (example Fig. 1). Formally, the output of i^{th} layer having n_i neurons, $h_i \in \mathbb{R}_i^n$ is given as:

$$h_i = \psi_i(W_i^T h_{i-1} + b_i) \quad (5)$$

where $h_{i-1} \in \mathbb{R}^{n_{i-1}}$ is output from $(i-1)^{th}$ -layer, $W_i \in \mathbb{R}^{n_{i-1} \times n_i}$, $b_i \in \mathbb{R}^{n_i}$ are weights and biases, and ψ_i is activation function for i^{th} -layer. Here, $i \in \{1, 2, \dots, l\}$ for l -layered feed-forward network and h_0 denotes input to the network.

3.2 Convolutional neural network (CNN)

CNN is a special type of feed-forward network which commonly consists of convolution layer, application of non-linear activation function, and followed by pooling-layer. Let us see them one-by-one:

Convolution Operation ConvNets derive their name from the ‘‘Convolution Operation’’. The Convolution in case of ConvNet is to extract features from the input images. Convolution preserves the spatial relationships between pixels by learning image features using patches of input data.

Every image can be considered as a matrix of the pixel values where values can be real numbers indicating the colour of the pixel (generally from 0-256). This 2D input is convolved with weight matrices that are learned, called *filters* or *kernels*. Intuitively, in convolution operation, the filter slides over the input, obtains element-wise products, and sums them. Formally, for a given input $X \in \mathbb{R}^{d_h \times d_w}$ and filter $F \in \mathbb{R}^{d_f \times d_f}$, result is given as:

$$R_{m,n} = \sum_i \sum_j X_{(m-i),(n-j)} F_{i,j} \quad (6)$$

where $R \in \mathbb{R}^{d_{c_h} \times d_{c_w}}$, $d_{c_h} = d_h - f + 1$, $d_{c_w} = d_w - f + 1$.

These kernels are also called feature detectors that go over the every single part of the image and produce features which indicates presence or absence of certain characteristics. Then result obtained from the convolution operation are called as *feature maps*, or *Convolved Feature* or *Activation Map*. This means if we employ k kernels, we are going to see k feature maps corresponding to each kernel for a given image. In general multiple kernels are used in the network which is indicated with the depth of the network.

It is important to note that when the kernel is moved over pixels of the image, it can be done in different ways. It is called stride of the kernels. When the stride is 1 then we move the filters one pixel at a time. When the stride is

2, then the filters jump 2 pixels at a time as we slide them around. Having a larger stride will produce smaller feature maps.

Sometimes, it is convenient to pad the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix. A nice feature of zero paddings is that it allows us to control the size of the feature maps. Adding zero-padding is also called *wide convolution*, and not using zero-padding would be a *narrow convolution*.

Non-linear Activation An additional operation called ReLU has been used after every Convolution operation. ReLU is an element-wise operation and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our ConvNet since most of the real-world data would be non-linear. Convolution is a linear operation— element-wise matrix multiplication and addition—so we account for non-linearity by introducing a non-linear function like ReLU.

Pooling-layers Convolutional layers are often interweaved with pooling layers. In particular, there is a kind of layer called a *max-pooling* layer that is extremely popular. Often, from a high level perspective, we don't care about the precise point in time a feature is present. A max-pooling layer takes the maximum of features over small blocks of a previous layer. The output tells us if a feature was present in a region of the previous layer, but not precisely where.

Max-pooling layers kind of “zoom out”. They allow later convolutional layers to work on larger sections of the data, because a small patch after the pooling layer corresponds to a much larger patch before it. They also make us invariant to some very small transformations of the data. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.

Advantages

- The number of parameters to be learned do not depend on the number of features in the input. This is a huge improvement over the feed-forward network where suppose we have n features then at-least we have to learn matrix of $n \times k$ where k is output dimension. In convolution network, the number of parameters to be learnt depend on the size of the kernel and number. This also leads to computation minimization.
- This also leads to weight sharing as the learned kernel is applied all over the image. We do not learn new kernel for each patch of the image.
- Because the kernels are applied independent of each other the calculation at lest upto some point can be parallelized. This is in contrast to RNNs where because of sequential nature of the data parallelization is impossible.

Drawbacks

- A lot of training data is needed for the CNN to be effective. CNNs tend to be much slower because of operations like maxpool. In case the convolutional neural network is made up of multiple layers, the training process could take a particularly long time if the computer does not have a good GPU.
- The main component of a CNN is a convolutional layer. Its job is to detect important features in the image pixels. Layers that are deeper (closer to the input) will learn to detect simple features such as edges and color gradients, whereas higher layers will combine simple features into more complex features. Finally, dense layers at the top of the network will combine very high level features and produce classification predictions.

In a CNN, all low-level details are sent to all the higher level neurons. These neurons then perform further convolutions to check whether certain features are present. This is done by striding the receptive field and then replicating the knowledge across all the different neurons

CNN do not encode the position and orientation of the object into their predictions. They completely lose all their internal data about the pose and the orientation of the object and they route all the information to the same neurons that may not be able to deal with this kind of information. A CNN makes predictions by looking at an image and then checking to see if certain components are present in that image or not. If they are, then it classifies that image accordingly.

- While CNNs are translation-invariant, they are generally bad at handling rotation and scale-invariance without explicit data augmentation.

3.3 Recurrent neural network (RNN)

In a traditional neural network we assume that all inputs are independent of each other. But for many tasks the idea seemed very primitive. Unlike feed-forward networks, RNNs have loops. Because of this feedback mechanism, the output depends on the current as well as the previous input which also leads to a sequential behaviour where different states produce different outputs.

Sequence Modelling is the task of predicting what will be the next output depending on the previous sequence seen so far. The core idea is to get the probability of the occurrence of the sequence and select the next output which can maximize the probability. Another important deviation from the other types of inputs where the size is fixed, here the sequences can be of variable sizes. For example consider words in sentences, they can be of variable sizes and we have to predict the next word from the previously seen words: language modelling.

For these kinds of sequence modelling tasks, it is important to consider previous inputs. Feed-forward networks and Convolution Networks do not possess

such capabilities. Moreover, these models also require fixed input sizes. Suppose we forcefully try to model such tasks with FFN, we will encounter different problems.

Suppose we use a simple MLP network for the language modelling task. Usually, each entry of the network will have a vector of particular dimension. Let each word is represented by a vector of 10k dimension. We have 5 words here. So our total input becomes 50k dimension. By the above network we still preserve sequence information. But there are many limitations for implementing this idea. Suppose we have another sentence like “This plane has some limitations in landing but landed safely”. This sentence has 10 words and the input size becomes $10 \times 10k = 100k$ dimension. Similarly you have a wide range sentences with variable word counts. In the above model, however, the network is capable of handling a 50k dimension as the model is trained with a 5 word sentence. Any test sentences with further word counts may not yield the desired results when testing the setup.

As an argument for the above said limitation we can use the longest sentence as an input vector. If we do so, our network will become infinitely large and will have billions and billions of weights making it another limitation. The solution to this problem is to use Recurrent neural networks.

Thus we will encounter problem of considering previous inputs and variable size of inputs. Both these problems are solved with Recurrent Neural Networks which considers previous inputs as well as can handle variable size inputs. These models are good for solving time series related data. For solving such problems various RNN architectures can be used as shown in Figure 2.

There are many variants of RNNs: Long Short-term Memory (LSTM), Gated Recurrent Unit (GRU), etc. We present a simple RNN which is extended to produce these more advanced variants.

$$h_t = \psi_h(W_h^T x_t + U_h^T h_{t-1} + b_h) \quad (7)$$

$$y_t = \psi_y(W_y^T h_t + b_y) \quad (8)$$

where $h_t, h_{t-1} \in \mathbb{R}^{n_h}$ are hidden states at $t, t-1$, respectively, $x_t \in \mathbb{R}^{n_x}$ and $y_t \in \mathbb{R}^{n_y}$ are input and output at time t , $W_h \in \mathbb{R}^{n_x \times n_h}$, $W_y \in \mathbb{R}^{n_h \times n_y}$, $U_h \in \mathbb{R}^{n_h \times n_h}$ are weight matrices, $b_h \in \mathbb{R}^{n_h}$, $b_y \in \mathbb{R}^{n_y}$ are biases, and ψ_h, ψ_y are activation functions.

The vanilla RNN captures only previous inputs, sometimes it is important to capture future inputs as well. For these kind of problems another RNN is trained in reverse direction. This Bidirectional RNN is shown in Figure 3. Also, there can be multiple layers of the RNNs connected to each other that is called a deep RNN which is also shown in Figure 3.

Backpropagation Through Time Applying backpropagation in RNNs is called Backpropagation through time. This procedure requires us to expand (or unroll) the computational graph of an RNN one sequence step at a time. The unrolled RNN is essentially a feedforward neural network with the special

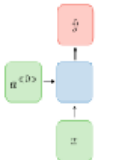
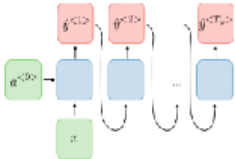
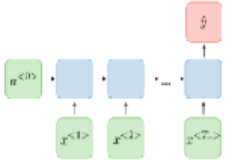
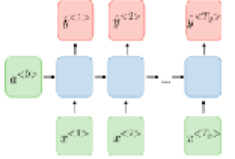
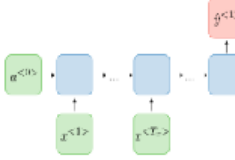
Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification
Many-to-many $T_x = T_y$		Name entity recognition
Many-to-many $T_x \neq T_y$		Machine translation

Figure 2: Different RNN architectures for different tasks.

property that the same parameters are repeated throughout the unrolled network, appearing at each sequence step. Then, just as in any feedforward neural network, we can apply the chain rule, backpropagating gradients through the unrolled net. The gradient with respect to each parameter must be summed across all places that the parameter occurs in the unrolled net. Handling such weight tying should be familiar from our chapters on convolutional neural networks.

Complications arise because sequences can be rather long. It is not unusual to work with text sequences consisting of over a thousand tokens. Note that this poses problems both from a computational (too much memory) and optimization (numerical instability) standpoint. Inputs from the first step passes through over 1000 matrix products before arriving at the output, and another 1000 matrix

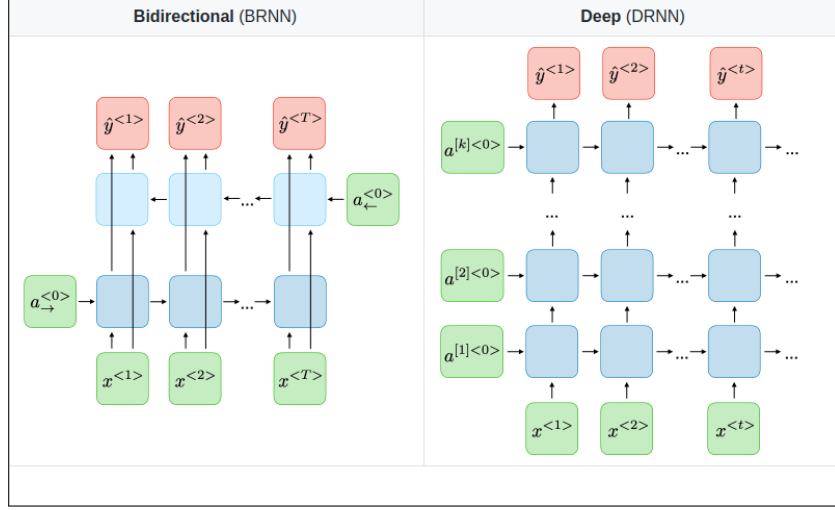


Figure 3: Variants of RNN.

products are required to compute the gradient. We now analyze what can go wrong and how to address it in practice.

We start with a simplified model of how an RNN works. This model ignores details about the specifics of the hidden state and how it is updated. The mathematical notation here does not explicitly distinguish scalars, vectors, and matrices. We are just trying to develop some intuition. In this simplified model, we denote h_t as the hidden state, x_t as input, and o_t as output at time step. The input and the hidden state can be concatenated to before being multiplied by one weight variable in the hidden layer. Thus, we use w_h and w_o to indicate the weights of the hidden layer and the output layer, respectively. As a result, the hidden states and outputs at each time steps are

$$h_t = f(x_t, h_{t-1}, w_h) \quad (9)$$

$$o_t = g(h_t, w_o) \quad (10)$$

where f and g are transformations of the hidden layer and the output layer, respectively. Hence, we have a chain of values $\{\dots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \dots\}$ that depend on each other via recurrent computation. The forward propagation is fairly straightforward. All we need is to loop through the triples (x_t, h_t, o_t) one time step at a time. The discrepancy between output o_t and the desired target y_t is then evaluated by an objective function across all the T time steps as

$$\mathcal{L}(x_1, \dots, x_T, y_1, \dots, y_T, w_h, w_o) = \frac{1}{T} \sum_{t=1}^T l(y_t, o_t) \quad (11)$$

For backpropagation, matters are a bit trickier, especially when we compute the gradients with regard to the parameters w_h of the objective function \mathcal{L} . To be specific, by the chain rule,

$$\frac{\partial \mathcal{L}}{\partial w_h} = \frac{1}{T} \sum_{t=1}^T \frac{\partial l(y_t, o_t)}{\partial w_h} \quad (12)$$

$$= \frac{1}{T} \sum_{t=1}^T \frac{\partial l(y_t, o_t)}{\partial o_t} \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial w_h} \quad (13)$$

The first and the second factors of the product in Eq. 13 are easy to compute as they involve the terms from the same time-step t . But the third factor $\frac{\partial h_t}{\partial w_h}$ is where things get tricky, since we need to recurrently compute the effect of the parameter w_h on h_t . This difficulty arises because h_t also depends on parameters w_h with the inputs from the previous time-step. According to the recurrent computation in Eq. 10, h_t depends on both h_{t-1} and w_h , where computation of h_{t-1} also depends on w_h . Thus, evaluating the total derivate of h_t with respect to w_h using the chain rule yields

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h} \quad (14)$$

We can remove the recurrent computation in Eq. 14 with

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^T \left(\prod_{j=i+1}^t \frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}} \right) \frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h} \quad (15)$$

While we can use the chain rule to compute recursively, this chain can get very long whenever t is large. Let's discuss a number of strategies for dealing with this problem.

- **Full Computation:** One idea might be to compute the full sum in Eq. 15. However, this is very slow and gradients can blow up, since subtle changes in the initial conditions can potentially affect the outcome a lot. That is, we could see things similar to the butterfly effect, where minimal changes in the initial conditions lead to disproportionate changes in the outcome. This is generally undesirable. After all, we are looking for robust estimators that generalize well. Hence this strategy is almost never used in practice.
- **Truncating Time Steps:** Alternatively, we can truncate the sum in Eq. 15 after τ steps. This leads to an approximation of the true gradient, simply by terminating the sum at $\frac{\partial h_{t-\tau}}{\partial w_h}$. In practice this works quite well. It is what is commonly referred to as truncated backpropagation through time. One of the consequences of this is that the model focuses primarily on short-term influence rather than long-term consequences. This is

actually desirable, since it biases the estimate towards simpler and more stable models.

To summarize:

- Backpropagation through time is merely an application of backpropagation to sequence models with a hidden state.
- Truncation is needed for computational convenience and numerical stability, such as regular truncation and randomized truncation.
- High powers of matrices can lead to divergent or vanishing eigenvalues. This manifests itself in the form of exploding or vanishing gradients.
- For efficient computation, intermediate values are cached during backpropagation through time.

Vanishing Gradient Problem As we have seen in the previous paragraph, the backpropagation through time is the way to calculate gradients with respect to parameters of RNN. This requires unrolling of the RNN through time and then calculating the derivative terms with the chain rule.

From the Eq. 15, we can see that the derivative with respect to the terms which are away from the current time step are going to be very small compared to the derivatives at the current step. This can lead to the vanishing gradient problem where the gradients become so small that the model does not learn from the sequences which have happened far in the past. This can be a major challenge in the training of the RNNs.

The last expression tends to vanish when k is large, this is due to the derivative of the tanh activation function which is smaller than 1. The product of derivatives can also explode if the weights w_h are large enough to overpower the smaller tanh derivative, this is known as the exploding gradient problem.

The Problem of Long-Term Dependencies One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful. But can they? It depends.

RNNs suffer from the problem of vanishing gradients, which hampers learning of long data sequences. The gradients carry information used in the RNN parameter update and when the gradient becomes smaller and smaller, the parameter updates become insignificant which means no real learning is done.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in “the clouds are in the sky,” we don’t need any further context – it’s pretty obvious the next word is going to be sky. In such cases, where the gap between the

relevant information and the place that it's needed is small, RNNs can learn to use the past information.

But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent French." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

Advantages

- The important architectural advantage is that it can consider the previous inputs while giving the current output. This is a major difference from the previous feed-forward architectures. Consider that FFN and CNN can accept input as only the current input, concretely suppose we are working on language modelling these models can take only current word while forgetting the previous words.
- It can also work with variable size of the input, this means that the sequence which we are considering can be of any length.
- Another advantage is that the weights are shared across the time steps which means there is little overhead of learning more parameters. This again means that the number of parameters are not going to increase as the input size increases.

Drawbacks

- These models suffer the vanishing gradient problem.
- There is a limitation on the length of sequence they can handle. In practice these models fail to capture long-term dependencies.
- As the computation is sequential, there is no chance of parallelization which means that the computation can be slower. This occurs mainly because unless and until we compute the previous time-step we can not process the next time-step.
- As these models can consider previous inputs but do not have capabilities to capture the information from future. This is solved with the use of employing another RNN network in the reverse direction and the whole setup is called Bidirectional RNNs.

3.3.1 LSTM

Long Short Term Memory networks (LSTMs) are a special kind of RNN, capable of learning long-term dependencies. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behaviour, not something they struggle to learn! The standard LSTM cell is shown in Figure 4.

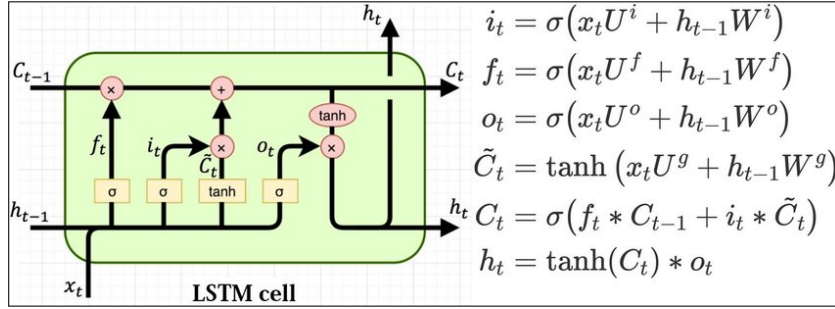


Figure 4: LSTM cell.

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”. An LSTM has three of these gates, to protect and control the cell state.

Different Components of LSTM

- **Which previous information to remove:** The first step in our LSTM is to decide what information we are going to throw away from the cell state. This decision is made by a sigmoid layer called the *forget gate*. It looks at h_{t-1} and x_t , and outputs a number between 0 and 1 for each number in the cell state C_{t-1} , where 1 means we store all the previous information whereas 0 means we remove the previous information.

$$f_t = \sigma(\mathbf{W}_f[h_{t-1}, x_t] + \mathbf{b}_f) \quad (16)$$

where σ denotes sigmoid function.

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

- **Which information to store:** The next step is to decide what new information we are going to store in the cell state. This has two parts. First, a sigmoid layer called the *input gate* decides which values we will update. Next, a *tanh* layer creates a vector of new candidate values, \tilde{C}_t , that could be added to the state. In the next step, we will combine these two to create an update to the state.

In the example of our language model, we would want to add the gender of the new subject to the cell state, to replace the old one we are forgetting.

$$i_t = \sigma(\mathbf{W}_i[h_{t-1}, x_t] + \mathbf{b}_i) \quad (17)$$

$$\tilde{C}_t = \tanh(\mathbf{W}_C[h_{t-1}, x_t] + \mathbf{b}_C) \quad (18)$$

- **Getting current state:** It is now time to update the old cell state, C_{t-1} , into the new cell state C_t . We multiply the old state by f_t , forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate value, scaled by how much we decided to update each state value.

In the case of the language model, this is where we would actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad (19)$$

- **Current Output:** Finally, we need to decide what we are going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we are going to output. Then, we put the cell state through *tanh* (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

$$o_t = \sigma(\mathbf{W}_o[h_{t-1}, x_t] + \mathbf{b}_o) \quad (20)$$

$$h_t = o_t * \tanh(C_t) \quad (21)$$

- **Consolidation**

$$f_t = \sigma(\mathbf{W}_f[h_{t-1}, x_t] + \mathbf{b}_f) \quad \dots \text{forget gate} \quad (22)$$

$$i_t = \sigma(\mathbf{W}_i[h_{t-1}, x_t] + \mathbf{b}_i) \quad \dots \text{input gate} \quad (23)$$

$$\tilde{C}_t = \tanh(\mathbf{W}_C[h_{t-1}, x_t] + \mathbf{b}_C) \quad (24)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad \dots \text{cell state} \quad (25)$$

$$o_t = \sigma(\mathbf{W}_o[h_{t-1}, x_t] + \mathbf{b}_o) \quad \dots \text{output} \quad (26)$$

$$h_t = o_t * \tanh(C_t) \quad \dots \text{hidden state} \quad (27)$$

Backpropagation through time in LSTMs As in the RNN model, our LSTM network outputs a prediction vector h_t on the t^{th} time step. The knowledge encoded in the state vectors C_t captures long-term dependencies and relations in the sequential data. We compute the gradient used to update the network parameters, the computation is done over T time steps.

The product term in the gradient causes it to vanish. For LSTM the product term comes out to be

$$\prod \frac{\partial C_t}{\partial C_{t-1}} \quad (28)$$

Recall that C_t value is obtained as

$$C_t = C_{t-1} \otimes f_t \oplus i_t \otimes \tilde{C}_t \quad (29)$$

Then the derivative can be obtained as:

$$\frac{\partial C_t}{\partial C_{t-1}} = \frac{\partial C_{t-1} \otimes f_t \oplus i_t \otimes \tilde{C}_t}{\partial C_{t-1}} \quad (30)$$

$$= \frac{\partial C_{t-1} \otimes f_t}{\partial C_{t-1}} + \frac{\partial i_t \otimes \tilde{C}_t}{\partial C_{t-1}} \quad (31)$$

$$= \frac{\partial f_t}{\partial C_{t-1}} \cdot C_{t-1} + \frac{\partial C_{t-1}}{\partial C_{t-1}} \cdot f_t + \frac{\partial i_t}{\partial C_{t-1}} \cdot \tilde{C}_t + \frac{\partial \tilde{C}_t}{\partial C_{t-1}} \cdot i_t \quad (32)$$

$$= \frac{\partial f_t}{\partial C_{t-1}} \cdot C_{t-1} + f_t + \frac{\partial i_t}{\partial C_{t-1}} \cdot \tilde{C}_t + \frac{\partial \tilde{C}_t}{\partial C_{t-1}} \cdot i_t \quad (33)$$

Notice that the gradient contains the forget gate's vector of activations, which allows the network to better control the gradients values, at each time step, using suitable parameter updates of the forget gate. The presence of the forget gate's activations allows the LSTM to decide, at each time step, that certain information should not be forgotten and to update the model's parameters accordingly.

Another important property to notice is that the cell state gradient is an additive function made up from four elements. This additive property enables

better balancing of gradient values during backpropagation. The LSTM updates and balances the values of the four components making it more likely the additive expression does not vanish.

This additive property is different from the RNN case where the gradient contained a single element inside the product. In RNNs, the sum is made from expressions with a similar behaviour that are likely to all be in $[0,1]$ which causes vanishing gradients.

In LSTMs, however, the presence of the forget gate, along with the additive property of the cell state gradients, enables the network to update the parameter in such a way that the different sub gradients do not necessarily agree and behave in a similar manner, making it less likely that all of the gradients will vanish, or in other words, the series of functions does not converge to zero.

Summing up, we have seen that RNNs suffer from vanishing gradients and caused by long series of multiplications of small values, diminishing the gradients and causing the learning process to become degenerate. In an analogous way, RNNs suffer from exploding gradients affected from large gradient values and hampering the learning process.

LSTMs solve the problem using a unique additive gradient structure that includes direct access to the forget gate's activations, enabling the network to encourage desired behaviour from the error gradient using frequent gates update on every time step of the learning process.

Advantages

- Solves the problem of vanishing gradient, in turn this produces better long-term dependency association.
- Because of different gates, there are better capabilities of remembering important information while forgetting the less substantial information.

Drawbacks

- More parameters to learn compared to plain vanilla RNN.
- Still the computation of gradients is going to be slow as still parallelization is not possible.

3.4 Attention

Initially attention mechanism was proposed with encoder-decoder RNN architecture which address the sequence-to-sequence nature of machine translation where input sequences differ in length from output sequences. From a high-level, the model is comprised of two sub-models as shown in Figure 5:

1. Encoder: It is responsible for stepping through the input time steps and encoding the entire sequence into a fixed length vector called a context vector.

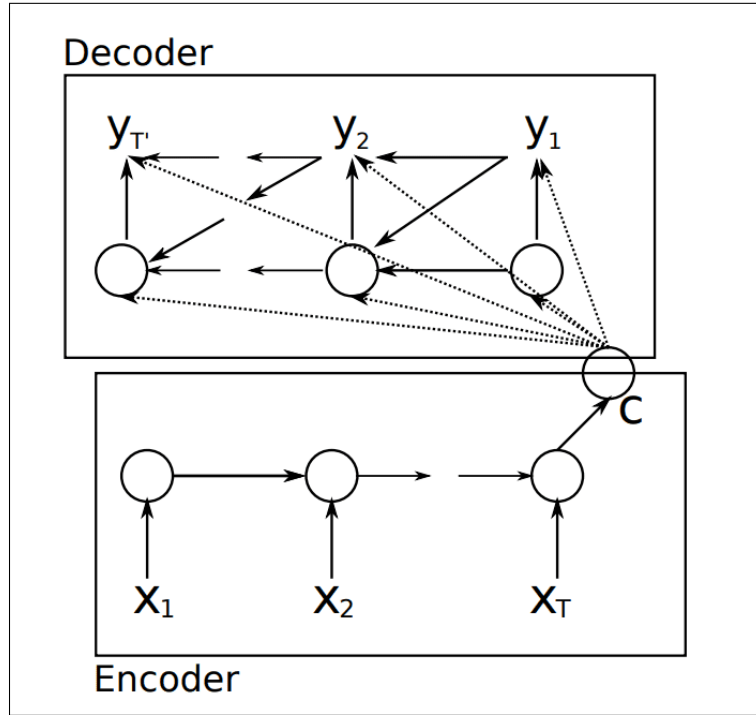


Figure 5: RNN Encoder-Decoder.

2. Decoder: It is responsible for stepping through the output time steps while reading from the context vector.

A potential issue with this encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector and have to decode the required information at each time-step in decoder. This may make it difficult for the neural network to cope with long sentences, especially those that are longer than the sentences in the training corpus. Attention is proposed as a solution to this limitation.

Instead of encoding the input sequence into a single fixed context vector, the attention model develops a context vector that is filtered specifically for each output time step. Attention searches for a set of positions in a source sentence where the most relevant information is concentrated at each time the proposed model generates a word in a translation. The model then predicts a target word based on the context vectors associated with these source positions and all the previous generated target words (as shown in Figure 6).

The context vector c_i depends on a sequence of annotations (h_1, \dots, h_T) to which an encoder maps the input sentence. Each annotation h_i contains information about the whole input sequence with a strong focus on the parts surrounding the i -th word of the input sequence.

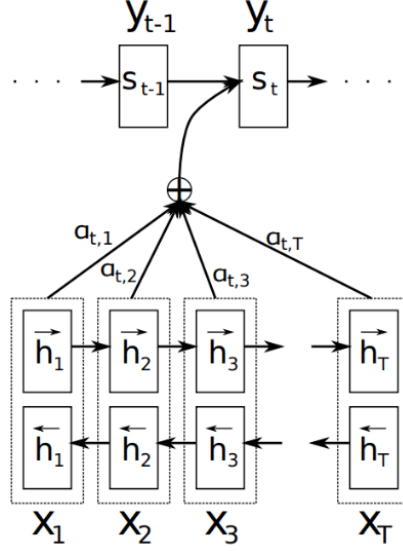


Figure 6: RNN with attention.

The context vector c_i is, then, computed as a weighted sum of the annotations h_i :

$$c_i = \sum_{j=1}^T \alpha_{ij} h_j \quad (34)$$

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp^{e_{ij}}}{\sum_{k=1}^T \exp^{e_{ik}}} \quad (35)$$

where $e_{ij} = a(s_{i-1}, h_j)$ is an attention model which scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden state s_{i-1} , just before emitting y_i , and the j -th annotation h_j of the input sentence. The attention model a is parametrized as a feedforward neural network which is jointly trained with all the other components of the proposed system.

There are two different types attention mechanisms which are conceptually similar but differ only how the query and key are obtained. When attention is performed on queries, keys and values generated from same embedding is called self attention, whereas if queries are generated from one embedding and keys and values generated from another embeddings is called cross attention.

3.5 Transformers

The Transformer architecture excels at handling text data which is inherently sequential. They take a text sequence as input and produce another text se-

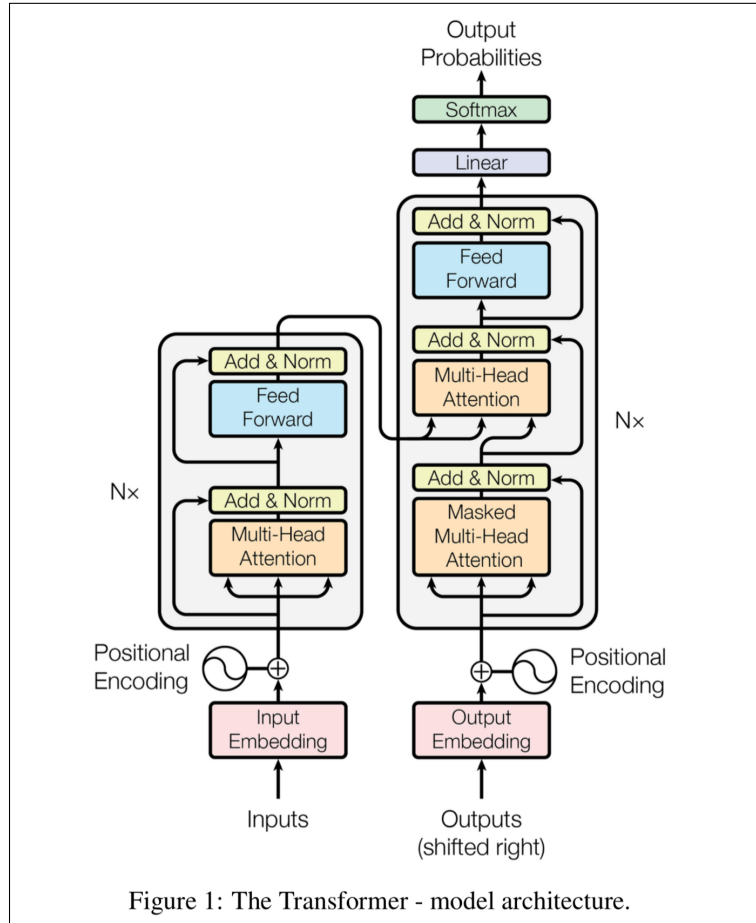


Figure 7: Transformer.

quence as output. eg. to translate an input English sentence to Spanish.

At its core, it contains a stack of Encoder layers and Decoder layers. To avoid confusion we will refer to the individual layer as an Encoder or a Decoder and will use Encoder stack or Decoder stack for a group of Encoder layers. The Encoder stack and the Decoder stack each have their corresponding Embedding layers for their respective inputs. Finally, there is an Output layer to generate the final output. The diagram of this internal structure is shown in Fig. 7.

All the Encoders are identical to one another. Similarly, all the Decoders are identical. The Encoder contains the all-important self-attention layer that computes the relationship between different words in the sequence, as well as a feed-forward layer. The Decoder is similar to Encoder which contains the self-attention layer and the feed-forward layer, and the differentiating part is of cross-attention between Encoder-Decoder.

Input Processing As an input to encoder and decoder are sentences which contain sequence of words and some special characters. Then on these sentences word-piece tokenizer is applied to get the tokens which are available in the vocabulary. If the word is not present in the vocabulary it is broken into constituent tokens which are present in the dictionary. Once the tokenization is done, the m -dimensional vector representation for each input *token* is obtained by summing their *token embeddings*, *sentence embeddings*, and *positional embeddings*.

The *token embeddings* are randomly initialized for each token in the dictionary whereas *sentence embeddings* for each token are determined based whether the token belongs to first or second sentence (denoted as A and B in the figure). In addition to that, BERT considers the position of the token in the sentence by learning positional embeddings. Because the attention mechanism is position invariant, meaning that the words may be shuffled and yet it does not impact the Transformer network's answer. That is why Transformer comes with yet another idea, a positional embedding layer which encodes the information about each word's position.

All these embeddings are of m -dimension so that they can be summed to produce vector $\mathbf{x}_i \in \mathbb{R}^m$ for i^{th} token. Let the m -dimensional vector representation of n tokens be arranged as row of the matrix $X \in \mathbb{R}^{n \times m}$ as $X = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{n-1}, \mathbf{x}_n]^T$. This matrix X is passed through l -layers of encoders to obtain the final representation. BERT is trained with two different values of l : BERT-base with $l = 12$ encoder layers and bigger BERT-large model with $l = 24$ encoders.

An encoder layer contains multiple attention heads as well as feed forward network as shown in Fig. 7. Similarly, decoder also contains these components. We look at these parts one by one so as to understand the model better.

Self-Attention Head : Both encoder and decoder consists of multiple attention heads, i.e., number of separate single attention heads. An attention head captures different interaction between all the tokens. Here, the number of attention heads is pre-decided, let that number be denoted as h .

The attention mechanism as a general convention follows a Query, Key, Value pattern. All three of these are words from the input sequence that are meant to operate with each other in a certain pattern. The query and key initially undergo certain operations, the output is then (usually) multiplied with the value. The overall operations are shown pictorially in Fig. 8.

Let us look at the mathematical operations happening in a single attention head to understand it better. Initially, the input matrix representation $X \in \mathbb{R}^{n \times m}$ is projected with three different matrices: $W^K \in \mathbb{R}^{m \times d_k}$, $W^V \in \mathbb{R}^{m \times d_v}$, $W^Q \in \mathbb{R}^{m \times d_k}$ to get *key*, *value*, and *query*. The dimensions are set to $d_k = d_v = m/h$ where m is input dimension. At each layer, each attention head in multi-head attention, learns different set of matrices, in turn yielding different key, query, and value. For instance, for i^{th} attention head, key, query,

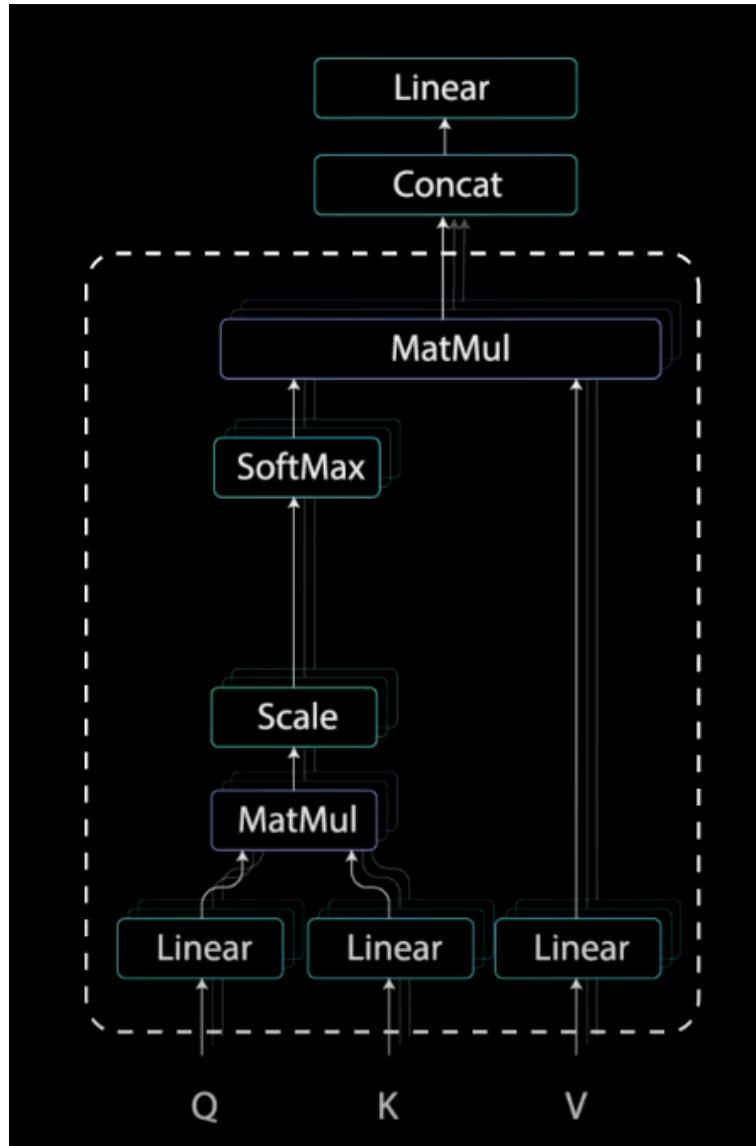


Figure 8: Attention operation with Query, Key, and Value.

and value matrices are obtained as:

$$K_i = XW_i^K \quad (36)$$

$$Q_i = XW_i^Q \quad (37)$$

$$V_i = XW_i^V \quad (38)$$

where $K_i \in \mathbb{R}^{n \times d_k}$, $V_i \in \mathbb{R}^{n \times d_v}$, $Q_i \in \mathbb{R}^{n \times d_k}$.

With the use of key and query matrices, $n \times n$ attention weight matrix between every token is calculated as:

$$A_i = \sigma\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) \quad (39)$$

where row-wise softmax function σ normalizes scores. This is the crucial part of the encoder architecture where different attention heads capture various interaction between input tokens. The score matrix determines how much focus should a word be put on other words. So each word will have a score that corresponds to other words in the time-step. The higher the score the more focus. This is how the queries are mapped to the keys.

Then, the scores get scaled down by getting divided by the square root of the dimension of query and key. This is to allow for more stable gradients, as multiplying values can have exploding effects.

This attention matrix $A_i \in \mathbb{R}^{n \times n}$ is used to weight the embeddings captured by the value matrix as:

$$H_i = A_i V_i \quad (40)$$

Multi Head Attention : So far we have seen the single attention head computations where the input is split into split the query, key, and value into N vectors before applying self-attention. The split vectors then go through the self-attention head process individually and each head produces an output vector. To make this a multi-headed attention computation, these outputs from different self-attention heads are concatenated into a single vector before going through the final linear layer. In theory, each head would learn something different therefore giving the encoder model more representation power.

Formally, the weighted embeddings output of each attention head is concatenated to produce the H as:

$$H = [H_1, H_2, \dots, H_h] \in \mathbb{R}^{n \times (h \cdot d_v)} \quad (41)$$

Finally, projection matrix $W_o \in \mathbb{R}^{(h \cdot d_v) \times m}$ is learned to get the output $M \in \mathbb{R}^{n \times m}$ dimension from multi-head attention same as input dimension with the following operation:

$$M = H W_o \quad (42)$$

Add & norm : The multi-headed attention output vector is added to the original input embeddings. This is called a residual connection. The output of the residual connection goes through a layer normalization.

The output from the multi-head attention or the feed-forward network is passed through this layer (green rectangle in Fig. ??). The relation between input I and output O obtained from this layer is given as:

$$O = \psi(I + \varphi(I)) \quad (43)$$

where ψ is layer normalization function, and φ is either feed-forward or multi-head attention output.

Feed-forward network : The normalized residual output gets projected through a pointwise feed-forward network for further processing. The pointwise feed-forward network is a couple of linear layers with a ReLU activation in between. The output of that is then again added to the input of the pointwise feed-forward network and further normalized.

This matrix M is passed through two layers as:

$$F = \max(0, MW_1 + b_1)W_2 + b_2 \quad (44)$$

where $W_1 \in \mathbb{R}^{m \times d_f}$, $W_2 \in \mathbb{R}^{d_f \times m}$ are weights, $b_1 \in \mathbb{R}^{n \times d_f}$, $b_2 \in \mathbb{R}^{n \times m}$ are biases, and $F \in \mathbb{R}^{n \times m}$.

The residual connections help the network train, by allowing gradients to flow through the networks directly. The layer normalizations are used to stabilize the network which results in substantially reducing the training time necessary. The pointwise feedforward layer is used to project the attention outputs potentially giving it a richer representation.

Encoder : As shown in Fig. 7, multiple encoders are stacked upon each other and the output of the underneath layer is passed to the above encoder. Finally, the output of the l^{th} layer encoder is used for predicting the masked tokens as well as next sentence in the structures like BERT which use only encoders.

Also, the output from the final encoder layer is given to all the decoder layers so as to get the cross-attention between the input sequence given to the encoder and the input sequence of the decoder. In the case of machine translation, it is intuitive to understand the relations between different token of the source language words and the target language tokens, which is achieved with this cross-attention. Specifically, in this cross-attention the inputs from encoders work as Query, Key and decoder output is considered as Value. Based on the attention score from query, key, the value is weighted.

Decoder The difference between encoder and decoder is that of masked multi-head attention between the decoder inputs and cross-attention between encoder output and decoder input. The masking of tokens which are occurring in the future is important for the inputs of decoder and as stated earlier, the cross-attention finds the interaction the decoder sequence and encoder sequence.

The decoder is autoregressive, it begins with a start token, and it takes in a list of previous outputs as inputs, as well as the encoder outputs that contain the attention information from the input. The decoder stops decoding when it generates a token as an output.

The beginning of the decoder is pretty much the same as the encoder. The input goes through an embedding layer and positional encoding layer to get positional embeddings. The positional embeddings get fed into the first multi-head attention layer which computes the attention scores for the decoder's input.

The first multi-headed attention layer of decoder operates slightly differently from the multi-head attention of encoder. As mentioned earlier, it masks tokens which are occurring in the future. Since the decoder is autoregressive and generates the sequence word by word, we need to prevent it from conditioning to future tokens. For example, for the sentence "I am fine", when computing attention scores on the word "am", you should not have access to the word fine, because that word is a future word that was generated after. The word am should only have access to itself and the words before it. This is true for all other words, where they can only attend to previous words.

We need a method to prevent computing attention scores for future words. This method is called masking. To prevent the decoder from looking at future tokens, you apply a look ahead mask. The mask is added before calculating the softmax, and after scaling the scores. The mask is a matrix that's the same size as the attention scores filled with values of 0s and $-\infty$. When you add the mask to the scaled attention scores, you get a matrix of the scores, with the top right triangle filled with $-\infty$ s.

The reason for the mask is because once you take the softmax of the masked scores, the negative infinities get zeroed out, leaving zero attention scores for future tokens.

The second multi-headed attention layer. For this layer, the encoder's outputs are the queries and the keys, and the first multi-headed attention layer outputs are the values. This process matches the encoder's input to the decoder's input, allowing the decoder to decide which encoder input is relevant to put a focus on. The output of the second multi-headed attention goes through a pointwise feedforward layer for further processing.

The output of the final pointwise feedforward layer goes through a final linear layer, that acts as a classifier. The classifier is as big as the number of classes you have. For example, if you have 10,000 classes for 10,000 words, the output of that classifier will be of size 10,000. The output of the classifier then gets fed into a softmax layer, which will produce probability scores between 0 and 1. We take the index of the highest probability score, and that equals our predicted word.

At the time of inference there is no input to the decoder as we want to predict the output corresponding to the input given to encoder. In the case of machine translation task, we want to get the translated sequence. The procedure is to give the whole sentence which needs to be translated to the encoder and then give <START> as an input to the decoder. The output of the decoder is appended to the <START> token and recursively given to the decoder until we

get the end-of-sequence token.

Advantages

- **No Vanishing Gradient:** Transformers do not face problem of vanishing gradients as seen in the RNN networks. This also makes them better at handling long sequences. This happens mainly because of the stacked attentions heads which are capable of capturing associations at each level and then at the next level building associations on top-of-that, this leads to robustness at handling longer sequences.
- The multiple attention-heads are good at capturing different associations between input tokens. This produces better representation power than RNNs/LSTMs.
- The computations can be achieved faster because of the parallelization achieved as the model is no longer sequential.

Drawbacks

- The time complexity at the inference time is quadratic in the size of input. This is worse compared to the RNN time complexity. Suppose we have n tokens in the input and k tokens in the output then the many-to-many encoder-decoder type architecture of RNN is going to take $n + k$ time to do the computations. With attention mechanism it is going to take nk time-complexity which worse than vanilla RNN. But for transformer, it is going to be $n^2 + k^2 + nk$ because self-attention of the n input tokens, then self-attention of the k output tokens and nk for the cross-attention between input and output. This is a major drawback compared to the RNNs.

3.6 Autoencoders

An autoencoder (AE) is used to learn efficient codings of unlabeled data (unsupervised learning). The encoding is validated and refined by attempting to regenerate the input from the encoding. The autoencoder learns a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore insignificant noise.

After the proposal of original AE a lot of variants have been proposed to tackle different objectives, aiming to force the learned representations to assume useful properties. Examples are regularized autoencoders: Sparse, Denoising and Contractive, which are effective in learning representations for subsequent classification tasks, and Variational autoencoders, with applications as generative models. Autoencoders are applied to many problems, including facial recognition, feature detection, anomaly detection and acquiring the meaning of words.

Formally, given a input vector $\mathbf{x} \in R^d$ the encoder learns latent representation as follows:

$$\mathbf{z} = \mathcal{F}_e(\mathbf{x}; \theta) \quad (45)$$

Then the decoder tries to reconstruct original vector \mathbf{x} from \mathbf{z} :

$$\hat{\mathbf{x}} = \mathcal{F}_d(\mathbf{z}; \phi) \quad (46)$$

The encoder and decoder parameters θ, ϕ are learned by optimizing following reconstruction loss:

$$\ell(\theta, \phi) = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 = \|\mathbf{x} - \mathcal{F}_d(\mathcal{F}_e(\mathbf{x}; \theta); \phi)\|^2 \quad (47)$$

The decoder tries to get back the original signal. The simplest way to perform this task perfectly would be to just duplicate the signal at encoder then the decoder can just show back the signal. If the code space \mathcal{Z} has dimension larger than, or equal to, the message space \mathcal{X} or the hidden units are given enough capacity, this type of AE is called *overcomplete*. Such an autoencoder can learn the identity function and become useless. However, experimental results found that overcomplete autoencoders might still learn useful features.

To suppress this behavior, the code space \mathcal{Z} usually has fewer dimensions than the message space \mathcal{X} . Such an autoencoder is called *undercomplete*. It can be interpreted as compressing the message, or reducing its dimensionality. In the ideal setting, the code dimension and the model capacity could be set on the basis of the complexity of the data distribution to be modeled. A standard way to do so is to add modifications to the basic autoencoder, to be detailed below.

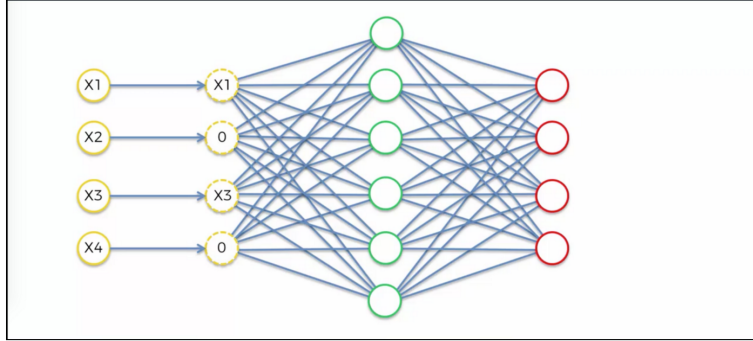


Figure 9: Denoising Autoencoder.

3.6.1 Denoising Autoencoders

In the case of overcomplete autoencoders, there is a risk of them learning an identity function and in turn becoming useless. Denoising Autoencoders solve

this problem by corrupting the input data on purpose. One of the simplest way is to randomly turn some of the input values to zero. In general, the percentage of input nodes which are being set to zero is about 50%. Other sources suggest a lower count, such as 30%. It depends on the amount of data and input nodes you have. There can be other approach of noising the input such as addition of Gaussian noise.

The loss function is similar where the model is trained to minimize the reconstructions loss. However, the input to which the output is compared is not the noisy but the original input. This makes sense as we are trying to make the model more robust where it can produce the original data even from the noisy data.

3.7 Variational Autoencoders

Generative modeling is a broad area of machine learning which deals with models of distributions $P(X)$, defined over datapoints \mathbf{x} in some potentially high-dimensional space \mathcal{X} . For instance, images are a popular kind of data for which we might create generative models. Each datapoint (image) has thousands or millions of dimensions (pixels), and the generative model’s job is to somehow capture the dependencies between pixels, e.g., that nearby pixels have similar color, and are organized into objects. Exactly what it means to “capture” these dependencies depends on exactly what we want to do with the model. One straightforward kind of generative model simply allows us to compute $P(X)$ numerically. In the case of images, \mathbf{x} values which look like real images should get high probability, whereas images that look like random noise should get low probability. However, models like this are not necessarily useful: knowing that one image is unlikely does not help us synthesize one that is likely.

Instead, one often cares about producing more examples that are like those already in a database, but not exactly the same. We could start with a database of raw images and synthesize new, unseen images. We might take in a database of 3D models of something like plants and produce more of them to fill a forest in a video game. We could take handwritten text and try to produce more handwritten text. Tools like this might actually be useful for graphic designers. We can formalize this setup by saying that we get examples X distributed according to some unknown distribution $P_{gt}(X)$, and our goal is to learn a model P which we can sample from, such that P is as similar as possible to P_{gt} .

3.7.1 Latent Variable Models

When training a generative model, the more complicated the dependencies between the dimensions, the more difficult the models are to train. Take, for example, the problem of generating images of handwritten characters. Say for simplicity that we only care about modeling the digits 0-9. If the left half of the character contains the left half of a 5, then the right half cannot contain the left half of a 0, or the character will very clearly not look like any real digit. Intuitively, it helps if the model first decides which character to generate before

it assigns a value to any specific pixel. This kind of decision is formally called a latent variable. That is, before our model draws anything, it first randomly samples a digit value z from the set $[0, \dots, 9]$, and then makes sure all the strokes match that character, z is called *latent* because given just a character produced by the model, we don't necessarily know which settings of the latent variables generated the character.

Before we can say that our model is representative of our dataset, we need to make sure that for every datapoint \mathbf{x} in the dataset, there is one (or many) settings of the latent variables which causes the model to generate something very similar to \mathbf{x} . Formally, say we have a vector of latent variables \mathbf{z} in a high-dimensional space \mathcal{Z} which we can easily sample according to some probability density function (PDF) $P(\mathbf{z})$ defined over \mathcal{Z} . Then, say we have a family of deterministic functions $f(\mathbf{z}; \theta)$, parameterized by a vector θ in some space Θ , where $f : \mathcal{Z} \times \Theta \rightarrow \mathcal{X}$, f is deterministic, but if \mathbf{z} is random and θ is fixed, then $f(\mathbf{z}; \theta)$ is a random variable in the space \mathcal{X} . We wish to optimize θ such that we can sample \mathbf{z} from $P(\mathbf{z})$ and, with high probability, $f(\mathbf{z}; \theta)$ will be like the \mathbf{x} 's in our dataset.

To make this notion precise mathematically, we are aiming maximize the probability of each \mathbf{x} in the training set under the entire generative process, according to:

$$P(X) = \int P(X|z; \theta) P(z) dz \quad (48)$$

Here, $f(\mathbf{z}; \theta)$ has been replaced by a distribution $P(X|z; \theta)$, which allows us to make the dependence of X on z explicit by using the law of total probability. The intuition behind this framework—called maximum likelihood—is that if the model is likely to produce training set samples, then it is also likely to produce similar samples, and unlikely to produce dissimilar ones. In VAEs, the choice of this output distribution is often Gaussian, i.e., $P(X|z; \theta) = \mathcal{N}(X|f(z; \theta), \sigma^2 * I)$. That is, it has mean $f(\mathbf{z}; \theta)$ and covariance equal to the identity matrix I times some scalar σ (which is a hyperparameter). This replacement is necessary to formalize the intuition that some needs to result in samples that are merely like X .

In general, and particularly early in training, our model will not produce outputs that are identical to any particular X . By having a Gaussian distribution, we can use gradient descent (or any other optimization technique) to increase $P(X)$ by making $f(\mathbf{z}; \theta)$ approach X for some z , i.e., gradually making the training data more likely under the generative model. This wouldn't be possible if $P(X|z)$ was a Dirac delta function, as it would be if we used $X = f(\mathbf{z}; \theta)$ deterministically! Note that the output distribution is not required to be Gaussian: for instance, if X is binary, then $P(X|z)$ might be a Bernoulli parameterized by $f(\mathbf{z}; \theta)$. The important property is simply that $P(X|z)$ can be computed, and is continuous in θ . From here onward, we will omit θ from $f(\mathbf{z}; \theta)$ to avoid clutter.

3.7.2 Mathematical Intuition

3.8 Generative Adversarial Network

A generative adversarial network (GAN) is a deep neural network framework which is able to learn from a set of training data and generate new data with the same characteristics as the training data. For example, a generative adversarial network trained on photographs of human faces can generate realistic-looking faces which are entirely fictitious.

Generative adversarial networks consist of two neural networks, the generator and the discriminator, which compete against each other. The generator is trained to produce fake data, and the discriminator is trained to distinguish the generator's fake data from real examples. If the generator produces fake data that the discriminator can easily recognize as implausible, such as an image that is clearly not a face, the generator is penalized. Over time, the generator learns to generate more plausible examples.

3.8.1 Architecture

A generative adversarial network is made up of two neural networks:

- *Generator* which learns to produce realistic fake data from a random seed. The fake examples produced by the generator are used as negative examples for training the discriminator.
- *Discriminator*, which learns to distinguish the fake data from realistic data. If the generator produces implausible results, the discriminator penalizes the generator.

The generator's fake examples, and the training set of real examples, are both fed randomly into the discriminator network. The discriminator does not know whether a particular input originated from the generator or from the training set. Initially, before training has begun, the generator's fake output is very easy for the discriminator to recognize.

Since the output of the generator is fed directly into the discriminator as input, this means that when the discriminator classifies an output of the generator, we can apply the backpropagation algorithm through the whole system and update the generator's weights. Over time, the generator's output becomes more realistic and the generator gets better at fooling the discriminator. Eventually, the generator's outputs are so realistic, that the discriminator is unable to distinguish them from the real examples.

3.8.2 Training of Discriminator and Generator

The discriminator is simply a binary classifier, ending with a suitable function such as the binary cross-entropy. The discriminator outputs a probability where the number indicates the discriminator's estimate of the probability of the input example being real or fake. The discriminator's input may come from two

sources: (1) the training set, such as real photos of faces, or real audio recordings, (2) the generator, such as generated synthetic faces, or fake audio recordings. While we are training the discriminator, we do not train the generator, but hold the generator's weights constant and use it to produce negative examples for the discriminator.

Discriminator Training process is same as the process for training any other kind of binary classifier:

- Pass some real examples, and some fake examples from the generator, into the discriminator as input.
- The discriminator classifies them into real and fake.
- Calculate the discriminator loss using a suitable function such as the cross-entropy loss.
- Update the discriminator's weights through backpropagation.

4 Regularization

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as regularization. As we will see there are a great many forms of regularization available to the deep learning practitioner. In fact, developing more effective regularization strategies has been one of the major research efforts in the field.

The no free lunch theorem implies that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems we ask the algorithm to solve, it performs better.

So far, the only method of modifying a learning algorithm that we have discussed concretely is to increase or decrease the model's representational capacity by adding or removing functions from the hypothesis space of solutions the learning algorithm is able to choose. We gave the specific example of increasing or decreasing the degree of a polynomial for a regression problem. The view we have described so far is oversimplified.

Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space. We can think of excluding a function from a hypothesis space as expressing an infinitely strong preference against that function.

In the weight decay ($\mathbf{w}^T \mathbf{w}$), we expressed our preference for linear functions defined with smaller weights explicitly, via an extra term in the criterion we minimize. There are many other ways of expressing preferences for different

solutions, both implicitly and explicitly. Together, these different approaches are known as regularization. Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Regularization is one of the central concerns of the field of machine learning, rivalled in its importance only by optimization. The no free lunch theorem has made it clear that there is no best machine learning algorithm, and, in particular, no best form of regularization. Instead we must choose a form of regularization that is well-suited to the particular task we want to solve. The philosophy of deep learning in general and this book in particular is that a very wide range of tasks (such as all of the intellectual tasks that people can do) may all be solved effectively using very general-purpose forms of regularization.

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias. In generalization and overfitting, we focus on three situations, where the model family being trained either (1) excluded the true data generating process—corresponding to underfitting and inducing bias, or (2) matched the true data generating process, or (3) included the generating process but also many other possible generating processes—the overfitting regime where variance rather than bias dominates the estimation error. The goal of regularization is to take a model from the third regime into the second regime.

4.1 Parameter Norm Penalties

Regularization has been used for decades prior to the advent of deep learning. Linear models such as linear regression and logistic regression allow simple, straightforward, and effective regularization strategies.

Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\Omega(\theta)$ to the objective function J . We denote the regularized objective function by \tilde{J} :

$$\tilde{J}(\theta; X, Y) = J(\theta; X, Y) + \alpha\Omega(\theta) \quad (49)$$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω , relative to the standard objective function J . Setting α to 0 results in no regularization. Larger values of α correspond to more regularization.

When our training algorithm minimizes the regularized objective function \tilde{J} it will decrease both the original objective J on the training data and some measure of the size of the parameters θ (or some subset of the parameters).

Different choices for the parameter norm Ω can result in different solutions being preferred. In this section, we discuss the effects of the various norms when used as penalties on the model parameters.

We note that for neural networks, we typically choose to use a parameter norm penalty Ω that penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized. The biases typically require less data to fit accurately than the weights. Each weight specifies how two variables interact. Fitting the weight well requires observing both variables in a variety of conditions. Each bias controls only a single variable. This means that we do not induce too much variance by leaving the biases unregularized. Also, regularizing the bias parameters can introduce a significant amount of underfitting. We therefore use the vector w to indicate all of the weights that should be affected by a norm penalty, while the vector θ denotes all of the parameters, including both w and the unregularized parameters.

4.1.1 L^2 Parameter Regularization

One of the simplest and most common kinds of parameter norm penalty: the L^2 parameter norm penalty commonly known as weight decay. This regularization strategy drives the weights closer to the origin (i.e. it really shrinks weights near to 0) by adding a regularization term $\Omega(\theta) = ||w||_2^2$ to the objective function. In other academic communities, L^2 regularization is also known as *ridge regression* or *Tikhonov regularization*.

We can gain some insight into the behaviour of weight decay regularization by studying the gradient of the regularized objective function. To simplify the presentation, we assume no bias parameter, so θ is just w . Such a model has the following total objective function:

$$\tilde{J}(w; X, Y) = J(w; X, Y) + \frac{\alpha}{2} w^T w \quad (50)$$

Gradient with respect to w , produces following equation:

$$\nabla_w \tilde{J}(w; X, Y) = \alpha w + \nabla_w J(w; X, Y) \quad (51)$$

To take a single gradient step to update the weights, we perform this update:

$$\begin{aligned} w &\leftarrow w - \epsilon(\alpha w + \nabla_w J(w; X, Y)) \\ w &\leftarrow (1 - \alpha\epsilon)w - \epsilon \nabla_w J(w; X, Y) \end{aligned}$$

We can see that the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update. This describes what happens in a single step.

4.1.2 L^1 Parameter Regularization

While L^2 weight decay is the most common form of weight decay, there are other ways to penalize the size of the model parameters. Another option is to use L^1 regularization. It is also known as Lasso Regression: Least Absolute Shrinkage and Selection Operator.

Formally, L^1 regularization on the model parameter w is defined as:

$$\Omega(w) = \|w\|_1 = \sum_i |w_i| \quad (52)$$

that is, as the sum of absolute values of the individual parameters. We will now discuss the effect of L^1 regularization on the simple linear regression model, with no bias parameter, that we studied in our analysis of L^2 regularization. In particular, we are interested in delineating the differences between L^1 and L^2 forms of regularization. As with L^2 weight decay, L^1 weight decay controls the strength of the regularization by scaling the penalty Ω using a positive hyperparameter α .

Thus, the regularized objective function $\tilde{J}(w; X, Y)$ is given by

$$\tilde{J}(w; X, Y) = \alpha \|w\|_1 + J(w; X, Y) \quad (53)$$

with the corresponding gradient (actually, sub-gradient):

$$\nabla_w \tilde{J}(w; X, Y) = \alpha \text{sign}(w) + \nabla_w J(w; X, Y) \quad (54)$$

where $\text{sign}(w)$ is simply the sign of w applied element-wise.

By inspecting the above equation, we can see immediately that the effect of L^1 regularization is quite different from that of L^2 regularization. Specifically, we can see that the regularization contribution to the gradient no longer scales linearly with each w_i ; instead it is a constant factor with a sign equal to $\text{sign}(w_i)$.

We can see from the above equation that the effect of L^1 regression is different than L^2 regression. L^1 regression drives weights to 0 thus producing sparsity in the model. It also means if we have a lot of features then L^1 regularization can act as a feature selection as opposed to L^2 regularization which shrinks all weights but do not make them 0. Therefore, L^1 regularization is the preferred choice when having a high number of features as it provides sparse solutions. Even, we obtain the computational advantage because features with zero coefficients can be avoided.

The main intuitive difference between the L^1 and L^2 regularization is that L^1 regularization tries to estimate the median of the data while the L^2 regularization tries to estimate the mean of the data to avoid overfitting. **What does it mean?**

Many regularization techniques can be interpreted as MAP Bayesian inferences. L^2 in particular is almost equivalent to MAP Bayesian inference with a Gaussian prior on the weights. In L^1 regularization, the penalty term used to penalize the cost function can be compared to the log-prior term that is maximized by MAP Bayesian inference when the prior is an isotropic Laplace Distribution over the real number dataset.

Why does this happen? Why L^1 penalty makes few weights 0 and L^2 shrinks weights?

In case of L^1 regularization, we are minimising sum of absolute values. When we think about it, we get the minimum value when the individual weights are 0. That is the only possibility and that is the only minimum we get with L^1

regularization. Thus, in case of L^1 regularization, the weights are reduced to 0, leading to a sparse solution.

Now, let us look at the L^2 regularization, the minimum value is obtained when the value of individual weights is 0. I was thinking there might be some difference but it seems both of them attain minimum when we attain the 0 value. It means that the whole previous discussion did not yield anything!!

4.2 Dataset Augmentation

The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. For some machine learning tasks, it is reasonably straightforward to create new fake data.

This approach is easiest for classification. A classifier needs to take a complicated, high dimensional input x and summarize it with a single category identity y . This means that the main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new (x, y) pairs easily just by transforming the x inputs in our training set. This approach is not as readily applicable to many other tasks. For example, it is difficult to generate new fake data for a density estimation task unless we have already solved the density estimation problem.

Dataset augmentation has been a particularly effective technique for a specific classification problem: object recognition. Images are high dimensional and include an enormous variety of factors of variation, many of which can be easily simulated. Operations like translating the training images a few pixels in each direction can often greatly improve generalization, even if the model has already been designed to be partially translation invariant by using the convolution and pooling techniques. Many other operations such as rotating the image or scaling the image have also proven quite effective.

Injecting noise in the input to a neural network can also be seen as a form of data augmentation. For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input. Neural networks prove not to be very robust to noise, however. One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs. Input noise injection is part of some unsupervised learning algorithms such as the denoising autoencoder. Noise injection also works when the noise is applied to the hidden units, which can be seen as doing dataset augmentation at multiple levels of abstraction.

When comparing machine learning benchmark results, it is important to take the effect of dataset augmentation into account. Often, hand-designed dataset augmentation schemes can dramatically reduce the generalization error of a machine learning technique. To compare the performance of one machine learning algorithm to another, it is necessary to perform controlled experiments. When comparing machine learning algorithm A and machine learning algorithm B, it is necessary to make sure that both algorithms were evaluated using the

same hand-designed dataset augmentation schemes. Suppose that algorithm A performs poorly with no dataset augmentation and algorithm B performs well when combined with numerous synthetic transformations of the input. In such a case it is likely the synthetic transformations caused the improved performance, rather than the use of machine learning algorithm B. Sometimes deciding whether an experiment has been properly controlled requires subjective judgement. For example, machine learning algorithms that inject noise into the input are performing a form of dataset augmentation. Usually, operations that are generally applicable (such as adding Gaussian noise to the input) are considered part of the machine learning algorithm, while operations that are specific to one application domain (such as randomly cropping an image) are considered to be separate pre-processing steps.

4.3 Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. This behaviour occurs very reliably. This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters.

The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. This procedure is specified more formally in algorithm 1. This strategy is known as early stopping. It is probably the most commonly used form of regularization in deep learning. Its popularity is due both to its effectiveness and its simplicity. One way to think of early stopping is as a very efficient hyperparameter selection algorithm. In this view, the number of training steps is just another hyperparameter.

Early stopping is a very unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. This means that it is easy to use early stopping without damaging the learning dynamics. This is in contrast to weight decay, where one must be careful not to use too much weight decay and trap the network in a bad local minimum corresponding to a solution with pathologically small weights.

Early stopping may be used either alone or in conjunction with other regularization strategies. Even when using regularization strategies that modify the objective function to encourage better generalization, it is rare for the best generalization to occur at a local minimum of the training objective.

Early stopping requires a validation set, which means some training data is not fed to the model. To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed. In the

Algorithm 1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

```

1: Let  $n$  be the number of steps between evaluations.
2: Let  $p$  be the “patience”, the number of times to observe worsening validation
   set error before giving up.
3: Let  $\theta_0$  be the initial parameters.
4:  $\theta \leftarrow \theta_0$ 
5:  $i \leftarrow 0$ 
6:  $j \leftarrow 0$ 
7:  $v \leftarrow \infty$ 
8:  $\theta^* \leftarrow \theta$ 
9:  $i^* \leftarrow i$ 
10: while  $j < p$  do
11:   Update  $\theta$  by running the training algorithm for  $n$  steps.
12:    $i \leftarrow i + n$ 
13:    $v' \leftarrow \text{ValidationSetError}(\theta)$ 
14:   if  $v' < v$  then
15:      $j \leftarrow 0$ 
16:      $v \leftarrow v'$ 
17:      $\theta^* \leftarrow \theta$ 
18:      $i^* \leftarrow i$ 
19:   else
20:      $j \leftarrow j + 1$ 
21: Best parameters are  $\theta^*$ , best number of training steps is  $i^*$ .

```

second, extra training step, all of the training data is included. There are two basic strategies one can use for this second training procedure.

4.4 Dropout

Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks. Bagging involves training multiple models, and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory.

Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks. Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network. In most modern neural networks, based on a series of affine transformations and nonlinearities,

we can effectively remove a unit from a network by multiplying its output value by zero. This procedure requires some slight modification for models such as radial basis function networks, which take the difference between the unit’s state and some reference value. Here, we present the dropout algorithm in terms of multiplication by zero for simplicity, but it can be trivially modified to work with other operations that remove a unit from the network.

Recall that to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i . Dropout aims to approximate this process, but with an exponentially large number of neural networks. Specifically, to train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent. Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network. The mask for each unit is sampled independently from all of the others. The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins. It is not a function of the current value of the model parameters or the input example. Typically, an input unit is included with probability 0.8 and a hidden unit is included with probability 0.5. We then run forward propagation, back-propagation, and the learning update as usual.

More formally, suppose that a mask vector μ specifies which units to include, and $J(\theta, \mu)$ defines the cost of the model defined by parameters θ and mask μ . Then dropout training consists in minimizing $E_{\mu} J(\theta, \mu)$. The expectation contains exponentially many terms but we can obtain an unbiased estimate of its gradient by sampling values of μ . Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent. In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

In the case of bagging, each model is trained to convergence on its respective training set. In the case of dropout, typically most models are not explicitly trained at all—usually, the model is large enough that it would be infeasible to sample all possible sub-networks within the lifetime of the universe. Instead, a tiny fraction of the possible sub-networks are each trained for a single step, and the parameter sharing causes the remaining sub-networks to arrive at good settings of the parameters. These are the only differences. Beyond these, dropout follows the bagging algorithm. For example, the training set encountered by each sub-network is indeed a subset of the original training set sampled with replacement.