

UNIT II: Introducing Deep Learning

Biological and Machine Vision, Human and Machine Language, Artificial Neural Networks, Training Deep Networks, Improving Deep Networks

.....

Chapter 1: Biological and Machine Vision

- Biological Vision
- Machine Vision
 - The Neocognitron
 - LeNet-5
 - The Traditional Machine Learning Approach
 - ImageNet and the ILSVRC
 - AlexNet
- TensorFlow PlayGround
- The *Quick, Draw!* Game

Chapter 2: Human and Machine Language

- Deep Learning for Natural Language Processing
 - Deep Learning Networks Learn Representations Automatically
 - A Brief History of Deep Learning for NLP
- Computational Representations of Language
 - One-Hot Representations of Words
 - Word Vectors
 - Word Vector Arithmetic
 - word2viz
 - Localist Versus Distributed Representations
- Elements of Natural Human Language
- Google Duplex

Chapter 3: Artificial Neural Networks

- The Input Layer
- Dense Layers
- A Hot Dog-Detecting Dense Network
 - Forward Propagation through the First Hidden Layer
 - Forward Propagation through Subsequent Layers
- The Softmax Layer of a Fast Food-Classifying Network
- Revisiting our Shallow Neural Network

Chapter 4: Training Deep Networks

- Cost Functions
 - Quadratic Cost
 - Saturated Neurons
 - Cross-Entropy Cost
- Optimization: Learning to Minimize Cost
 - Gradient Descent
 - Learning Rate
 - Batch Size and Stochastic Gradient Descent
 - Escaping the Local Minimum
- Backpropagation
- Tuning Hidden-Layer Count and Neuron Count
- An Intermediate Net in Keras

Chapter 5: Improving Deep Networks

- Weight Initialization
 - Xavier Glorot Distributions
- Unstable Gradients
 - Vanishing Gradients
 - Exploding Gradients
 - Batch Normalization
- Model Generalization — Avoiding Overfitting
 - L1 and L2 Regularization
 - Dropout
 - Data Augmentation
- Fancy Optimizers
 - Momentum
 - Nesterov Momentum
 - AdaGrad
 - AdaDelta and RMSProp
 - Adam
- A Deep Neural Network in Keras
- Regression
- TensorBoard

I. Biological and Machine Vision

Biological vision is the process by which animals see and process visual information. It is a complex system that involves the eyes, the brain, and the nervous system.

The eyes collect light from the environment and focus it onto the retina, a layer of light-sensitive cells at the back of the eye. The retina contains two types of photoreceptor cells: rods and cones. Rods are more sensitive to light than cones, but they do not provide color vision. Cones are responsible for color vision, but they are less sensitive to light than rods.

The photoreceptor cells in the retina convert light into electrical signals. These signals are then transmitted to the brain via the optic nerve. The brain processes the signals from the retina to create a visual image.

The visual system is not a simple camera. It is a complex system that is constantly adapting to the environment. The brain uses a variety of techniques to extract information from the visual image, including:

- **Edge detection:** The brain identifies edges in the visual image. Edges are important because they help to define objects.
- **Color detection:** The brain identifies the colors in the visual image. Color is important for recognizing objects and for understanding the environment.
- **Motion detection:** The brain identifies the motion of objects in the visual image. Motion is important for tracking objects and for understanding the environment.
- **Depth perception:** The brain uses a variety of cues to determine the depth of objects in the visual image. Depth perception is important for navigation and for interacting with the environment.

Biological vision is a remarkable feat of engineering. It allows animals to see and interact with the world in a very sophisticated way.

Here are some of the key differences between biological and machine vision:

- **Biological vision is analog, while machine vision is digital.** This means that biological vision works with continuous signals, while machine vision works with discrete signals.
- **Biological vision is adaptive, while machine vision is static.** This means that biological vision can change its response to the environment, while machine vision cannot.
- **Biological vision is robust to noise, while machine vision is not.** This means that biological vision can still function in the presence of noise, while machine vision can be easily fooled by noise.

Biological vision is still a mystery to scientists. However, the study of biological vision has helped to advance the field of machine vision. Machine vision systems are now able to perform many of the same tasks as biological vision, but they still have a long way to go before they can match the capabilities of biological vision.

2. MACHINE VISION

- We've been talking about the biological visual system because it's the inspiration for modern machine vision techniques called deep learning.
- Figure 1.8 shows a timeline of vision in biological organisms and machines.
- The top timeline shows the development of vision in trilobites and Hubel and Wiesel's 1959 discovery of the hierarchical nature of the primary visual cortex.
- The machine vision timeline is split into two tracks: deep learning (pink) and traditional machine learning (purple).
- Deep learning is our focus, and it's more powerful and revolutionary than traditional machine learning."

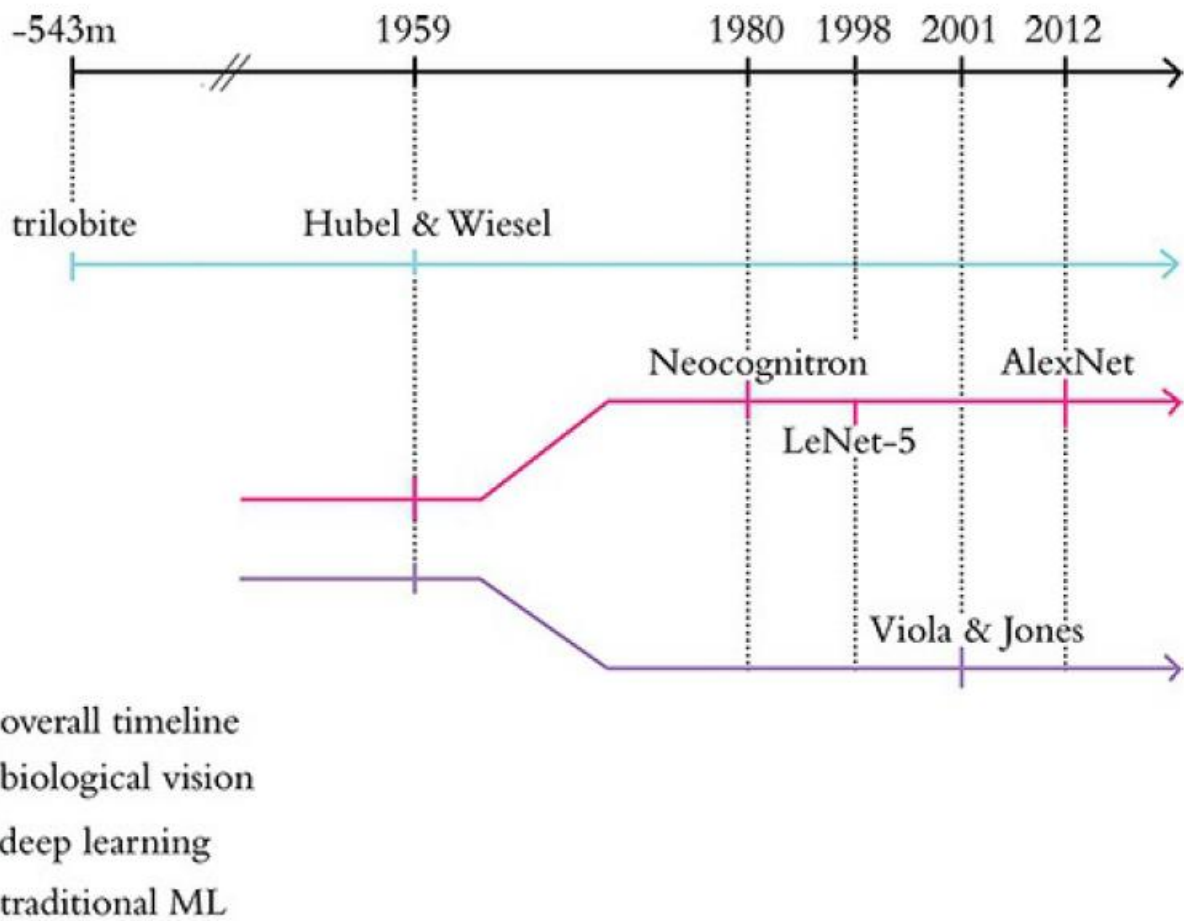


Figure 1.8 Abridged timeline of biological and machine vision, highlighting the key historical moments in the deep learning and traditional machine learning approaches to vision that are covered in this section

2.1. Neocognitron (నియోకాగ్నిట్రాన్)

The neocognitron is a hierarchical, multilayered artificial neural network. It has been used for Japanese handwritten character recognition and other pattern recognition tasks, and served as the inspiration for convolutional neural networks.

The neocognitron is inspired by the model proposed by Hubel & Wiesel in 1959. They found two types of cells in the visual primary cortex called simple cell and complex cell, and also proposed a cascading model of these two types of cells for use in pattern recognition tasks. The neocognitron is a natural extension of these cascading models.

The neocognitron (నియోకాగ్నిట్రాన్) consists of multiple layers of cells, each of which performs a specific function. The first layer of cells, called S-cells, detects edges and other local features in the input image. The second layer of cells, called C-cells, integrates the outputs of the S-cells to detect more complex features. The third layer of cells, called M-cells, integrates the outputs of the C-cells to detect objects.

The neocognitron is trained using a process called self-organization. In self-organization, the weights of the connections between the cells are adjusted so that the network learns to recognize a specific set of patterns. The neocognitron can be trained to recognize a variety of patterns, including handwritten characters, faces, and objects.

2.2.LeNet-5

LeNet-5 is a convolutional neural network (CNN) architecture proposed by Yann LeCun et al. in 1998. It was one of the first CNNs to achieve state-of-the-art results on the MNIST handwritten digit recognition dataset. LeNet-5 is a relatively simple CNN, but it is still a powerful architecture that can be used for a variety of image recognition tasks.

LeNet-5 consists of seven layers:

- 2 convolutional layers with 20 and 50 filters, respectively
- 2 max pooling layers with a pool size of 2x2
- 2 fully connected layers with 500 and 100 neurons, respectively
- A softmax layer with 10 outputs, one for each digit

The convolutional layers extract features from the input image. The max pooling layers reduce the size of the feature maps, while preserving the most important features. The fully connected layers classify the features into one of the 10 digits.

2.3.The Traditional Machine Learning Approach

Traditional machine learning is a type of machine learning that uses statistical methods to learn from data. It is a more general approach to machine learning than deep learning, and it can be used for a wider variety of tasks.

Traditional machine learning algorithms are typically divided into two categories: supervised learning and unsupervised learning.

- **Supervised learning** algorithms are trained on labeled data. This means that the data is already tagged with the correct output. The algorithm then learns to map the input data to the output data.
- **Unsupervised learning** algorithms are trained on unlabeled data. This means that the data does not have any tags. The algorithm then learns to find patterns in the data.

Traditional machine learning algorithms have been used for a variety of tasks, including:

- **Classification:** This is the task of assigning a label to an input data point. For example, a classification algorithm could be used to classify images as cats or dogs.
- **Regression:** This is the task of predicting a continuous value from an input data point. For example, a regression algorithm could be used to predict the price of a house based on its features.
- **Clustering:** This is the task of grouping data points together that are similar to each other. For example, a clustering algorithm could be used to group customers together based on their purchasing habits.

Traditional machine learning algorithms have been successful in many applications. However, they can be difficult to train and require a lot of data. They can also be sensitive to noise in the data.

Deep learning is a more recent approach to machine learning that uses artificial neural networks to learn from data. Deep learning algorithms have been shown to be more powerful than traditional machine learning algorithms for many tasks.

- **Traditional machine learning algorithms are typically shallow, while deep learning algorithms are deep.** This means that traditional machine learning algorithms have a few layers of neurons, while deep learning algorithms have many layers of neurons.
- **Traditional machine learning algorithms are typically supervised, while deep learning algorithms can be supervised or unsupervised.** This means that traditional machine learning algorithms require labeled data, while deep learning algorithms can learn from unlabeled data.

- **Traditional machine learning algorithms are typically designed by humans, while deep learning algorithms are often trained using reinforcement learning.** This means that traditional machine learning algorithms require human expertise to design, while deep learning algorithms can learn from data without any human intervention.

Deep learning has revolutionized the field of machine learning. It has been used to achieve state-of-the-art results on a variety of tasks, including image recognition, natural language processing, and speech recognition.

However, deep learning algorithms can be computationally expensive to train and require a lot of data. They can also be difficult to interpret, which can make it difficult to understand how they make decisions.

Traditional machine learning is still a valuable tool for machine learning. It is less computationally expensive than deep learning and it can be easier to interpret. Traditional machine learning is also a good choice for tasks where labeled data is scarce.

2.4. ImageNet and the ILSVRC

ImageNet is a large visual database designed for use in visual object recognition software research. It contains over 15 million labeled high-resolution images belonging to roughly 22,000 categories. The images were collected from the web and labeled by human labelers using Amazon's Mechanical Turk crowd-sourcing tool.

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual competition held since 2010 to test the progress of computer vision algorithms on the ImageNet dataset. The challenge tasks typically involve image classification, object detection, and scene recognition.

The ILSVRC has been instrumental in the development of deep learning for computer vision. In 2012, a deep convolutional neural network called AlexNet won the ILSVRC image classification task with a top-5 error rate of 16.4%. This was a significant improvement over previous methods, and it showed that deep learning could be used to achieve state-of-the-art results on image recognition tasks.

The ILSVRC has continued to drive progress in deep learning for computer vision. In recent years, the top-5 error rate on the ILSVRC image classification task has fallen below 2%. This is a remarkable achievement, and it shows that deep learning is now capable of accurately recognizing objects in images.

The ILSVRC has also been used to develop new deep learning architectures. For example, the VGGNet architecture, which won the ILSVRC image classification task in 2014, is now a popular choice for image recognition tasks.

The ILSVRC is a valuable resource for researchers in computer vision. It provides a large and challenging dataset for testing new algorithms, and it helps to drive progress in the field.

2.5. AlexNet

AlexNet is a convolutional neural network (CNN). It was one of the first CNNs to achieve state-of-the-art results on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) image classification task. AlexNet won the ILSVRC image classification task with a top-5 error rate of 16.4%, which was a significant improvement over previous method.

AlexNet consists of eight layers:

- 5 convolutional layers with 96, 256, 384, 384, and 256 filters, respectively
- 3 max pooling layers with a pool size of 2x2
- 3 fully connected layers with 4096, 4096, and 1000 neurons, respectively

The convolutional layers extract features from the input image. The max pooling layers reduce the size of the feature maps, while preserving the most important features. The fully connected layers classify the features into one of the 1000 object categories in the ImageNet dataset.

AlexNet uses a number of techniques that were innovative at the time, including:

- **ReLU activation function:** ReLU activation functions are now standard in deep learning, but they were not widely used before AlexNet. ReLU activation functions have several advantages over other activation functions, such as sigmoid and tanh functions. ReLU activation functions are faster to compute, and they do not suffer from the vanishing gradient problem.
- **Dropout:** Dropout is a technique for regularizing neural networks. Dropout randomly drops out some of the neurons in the network during training. This helps to prevent the network from overfitting the training data.
- **Data augmentation:** Data augmentation is a technique for artificially increasing the size of the training dataset. Data augmentation is done by applying random transformations to the training images, such as cropping, flipping, and rotating. This helps to prevent the network from overfitting the training data.

AlexNet was a breakthrough in the field of deep learning. It showed that deep learning could be used to achieve state-of-the-art results on image recognition tasks. AlexNet also popularized the use of convolutional neural networks for image recognition.

3. TensorFlow Playground

- ☞ TensorFlow Playground is a web application that allows you to experiment with neural networks. It is a great way to learn about neural networks and how they work.
- ☞ TensorFlow Playground is written in JavaScript and uses TensorFlow.js, a JavaScript library for TensorFlow. TensorFlow.js allows you to run TensorFlow models in the browser.
- ☞ To use TensorFlow Playground, you first need to create a new model. You can choose from a variety of different models, including classification models, regression models, and clustering models.
- ☞ Once you have created a model, you can start experimenting with it. You can add neurons to the model, change the activation functions, and adjust the learning rate.
- ☞ TensorFlow Playground also allows you to train your model on your own data. You can upload a CSV file with your data, and TensorFlow Playground will train the model on your data.
- ☞ Once your model is trained, you can use it to make predictions. You can drag and drop an image into the playground, and the model will predict the class of the image.
- ☞ TensorFlow Playground is a great tool for learning about neural networks. It is easy to use and it allows you to experiment with neural networks without having to write any code.

4. QUICK, DRAW!

To interactively experience a deep learning network carrying out a machine vision task in real time, navigate to quickdraw.withgoogle.com to play the Quick, Draw! game. Click Let's Draw! to begin playing the game. You will be prompted to draw an object, and a deep learning algorithm will guess what you sketch.

Chapter 2: Human and Machine Language

2.1. Q) Deep Learning for Natural Language Processing

- Deep Learning Networks Learn Representations Automatically
- A Brief History of Deep Learning for NLP

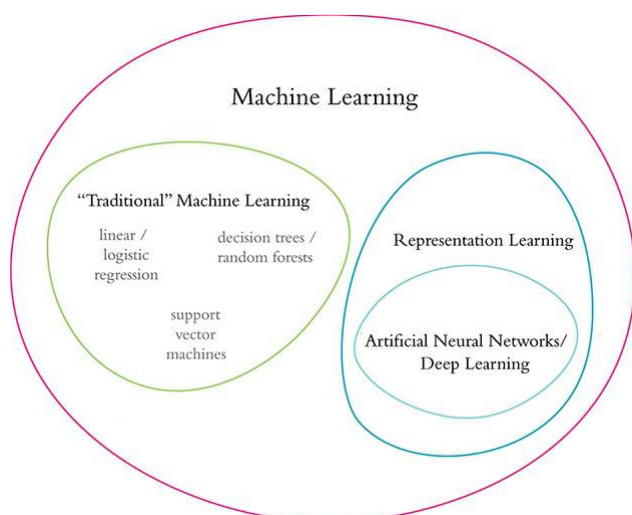
Deep learning is a type of machine learning that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain and are able to learn complex patterns from data. This makes them well-suited for natural language processing (NLP), which is the field of computer science that deals with the interaction between computers and human (natural) languages.

2.1 Deep Learning Networks Learn Representations Automatically

Deep learning can be defined as the layering of simple algorithms called artificial neurons into networks several layers deep.

The below Venn diagram, we show how deep learning resides within the machine learning family of representation learning approaches.

The representation learning family, which contemporary deep learning dominates, includes any techniques that learn features from data automatically. Indeed, we can use the terms “feature” and “representation” interchangeably.



The advantage of representation learning relative to traditional machine learning approaches is

Traditional ML typically works well because of clever, human-designed code that transforms raw data—whether it be images, audio of speech, or text from documents—into input features for machine learning algorithms (e.g., regression, random forest, or support vector machines) that are adept at weighting features but not particularly good at learning features from raw data directly.

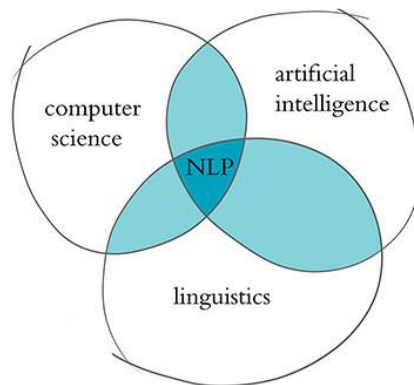
This manual creation of features is often a highly specialized task. For working with language data, for example, it might require graduate-level training in linguistics.

A primary benefit of deep learning is that it eases this requirement for subject-matter expertise. Instead of manually curating input features from raw data, one can feed the data directly into a deep learning model.

Over the course of many examples provided to the deep learning model, the artificial neurons of the first layer of the network learn how to represent simple abstractions of these data, while each successive layer learns to represent increasingly complex nonlinear abstractions on the layer that precedes it.

Natural Language Processing

Natural language processing is a field of research that sits at the intersection of computer science, linguistics, and artificial intelligence .



NLP involves taking the naturally spoken or naturally written language of humans—such as this sentence you are reading right now—and processing it with machines to automatically complete some task or to make a task easier for a human to do.

Examples of language use that do not fall under the umbrella of natural language could include code written in a software language or short strings of characters within a spreadsheet.

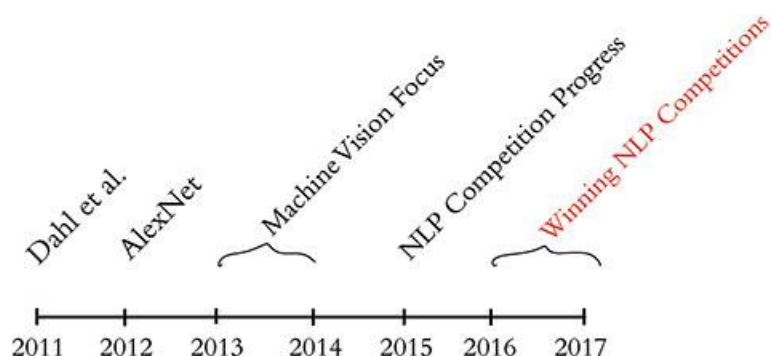
Examples of NLP in industry include:

1. *Classifying documents*: using the language within a document (e.g., an email, a Tweet, or a review of a film) to classify it into a particular category (e.g., high urgency, positive sentiment, or predicted direction of the price of a company's stock).
2. *Machine translation*: assisting language-translation firms with machine-generated suggestions from a source language (e.g., English) to a target language (e.g., German or Mandarin);
3. *Search engines*: autocompleting users' searches and predicting what information or website they're seeking.
4. *Speech recognition*: interpreting voice commands to provide information or act, as with virtual assistants like Amazon's Alexa, Apple's Siri, or Microsoft's Cortana.
5. *Chatbots*: carrying out a natural conversation for an extended period of time; though this is seldom done convincingly today, they are nevertheless helpful for relatively linear conversations on narrow topics such as the routine components of a firm's customer service phone calls.

Some of the easiest NLP applications to build are spell checkers, synonym suggesters, and keyword-search querying tools. These simple tasks can be fairly straightforwardly solved with deterministic, rules-based code using, say, reference dictionaries or thesauruses.

2.2. A Brief History of Deep Learning for NLP

The timeline, calls out recent milestones in the application of deep learning to NLP.



This timeline begins in 2011, when the University of Toronto computer scientist George Dahl and his colleagues at Microsoft Research revealed the first major breakthrough involving a deep learning algorithm applied to a large dataset.

This breakthrough happened to involve natural language data. Dahl and his team trained a deep neural network to recognize a substantial vocabulary of words from audio recordings of human speech.

The next landmark deep learning feat also came out of Toronto: AlexNet blowing the traditional machine learning competition out of the water in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

By 2015, the deep learning progress being made in machine vision began to spill over into NLP competitions such as those that assess the accuracy of machine translations from one language into another.

In 2016 and 2017, deep learning models entered into NLP competitions not only were more efficient than traditional machine learning models, but they also began outperforming them on accuracy.

2.2.Q) Computational Representations of Language

- One-Hot Representations of Words
- Word Vectors
- Word Vector Arithmetic
- word2viz
- Localist Versus Distributed Representations

Answer:

In order for deep learning models to process language, we have to supply that language to the model in a way that it can digest.

For all computer systems, this means a quantitative representation of language, such as a two-dimensional matrix of numerical values. Two popular methods for converting text into numbers are **one-hot encoding** and **word vectors**

2.2.1. one-hot encoding

One-hot encoding is the process of turning categorical factors into a numerical structure that machine learning algorithms can readily process. It functions by representing each category in a feature as a binary vector of 1s and 0s, with the vector's size equivalent to the number of potential categories.

For example, *if we have a feature with three categories (A, B, and C),*

each category can be represented as a binary vector of length three,

with the vector for category A being [1, 0, 0],

the vector for category B being [0, 1, 0], and the vector for category C being [0, 0, 1].

Why One-Hot Encoding is Used in NLP:

- One-hot encoding is used in NLP to encode categorical factors as binary vectors, such as words or part-of-speech identifiers.
- This approach is helpful because machine learning algorithms generally act on numerical data, so representing text data as numerical vectors are required for these algorithms to work.

- In a sentiment analysis assignment, for example, we might describe each word in a sentence as a one-hot encoded vector and then use these vectors as input to a neural network to forecast the sentiment of the sentence.

Example 1:

Suppose we have a small corpus of text that contains three sentences:

The quick brown fox jumped over the lazy dog.

She sells seashells by the seashore.

Peter Piper picked a peck of pickled peppers.

- Each word in these phrases should be represented as a single compressed vector. The first stage is to determine the categorical variable, which is the phrases' terms. The second stage is to count the number of distinct words in the sentences to calculate the number of potential groups. In this instance, there are 17 potential categories.
- The third stage is to make a binary vector for each of the categories. Because there are 17 potential groups, each binary vector will be 17 bytes long. For example, the binary vector for the word "quick" will be [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], with the 1s in the first and sixth places because "quick" is both the first and sixth group in the list of unique words.
- Finally, we use the binary vectors generated in step 3 to symbolize each word in the sentences as a one-hot encoded vector. For example, the one-hot encoded vector for the word "quick" in the first sentence is [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], and the one-hot encoded vector for the word "seashells" in the second sentence is [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0].

One-hot encoding is a simple and straightforward way to represent words, but it has some limitations. One limitation is that it does not take into account the context of the word. For example, the word "cat" can have different meanings depending on the sentence it is in. In the sentence "The cat sat on the mat", the word "cat" refers to a specific animal, but in the sentence "I like cats", the word "cat" refers to a general category of animals. One-hot encoding does not capture this difference in meaning.

Another limitation of one-hot encoding is that it can be computationally expensive to represent large vocabularies. For example, if the vocabulary contains 10,000 words, then each word would be represented by a 10,000-dimensional vector. This can be a problem for machine learning models that have to deal with large amounts of data.

Despite its limitations, one-hot encoding is a popular and effective way to represent words in natural language processing. It is simple to implement and can be used with a variety of machine learning models.

advantages

- It is a simple and straightforward way to represent words.
- It is easy to implement and can be used with a variety of machine learning models.
- It is a lossless encoding, meaning that no information about the word is lost.

disadvantages

- It does not take into account the context of the word.
- It can be computationally expensive to represent large vocabularies.
- It can lead to overfitting, especially when the vocabulary is large.

2.2.2. Word Vectors

Vector representations of words are the information-dense alternative to one-hot encodings of words. Whereas one-hot representations capture information about word location only, *word vectors* (also known as *word embeddings* or *vector-space embeddings*) capture information about word meaning as well as location.

Goal of Word Embeddings

- To reduce dimensionality
- To use a word to predict the words around it
- Inter word semantics must be captured

How are Word Embeddings used?

- They are used as input to machine learning models.
- Take the words —> Give their numeric representation —> Use in training or inference
- To represent or visualize any underlying patterns of usage in the corpus (It a collection of all the documents present in our dataset.) that was used to train them.

Two of the most popular techniques for converting natural language into word vectors are word2vec and GloVe

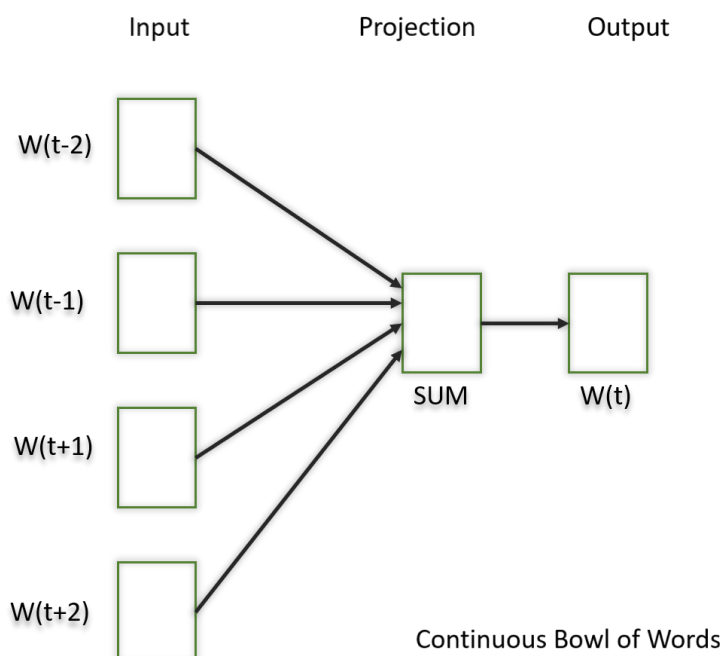
1) Word2Vec:

In Word2Vec every word is assigned a vector. We start with either a random vector or **one-hot vector**.

One-Hot vector: A representation where only one bit in a vector is 1. If there are 500 words in the corpus then the vector length will be 500. After assigning vectors to each word we take a window size and iterate through the entire corpus. While we do this there are two **neural embedding methods** which are used:

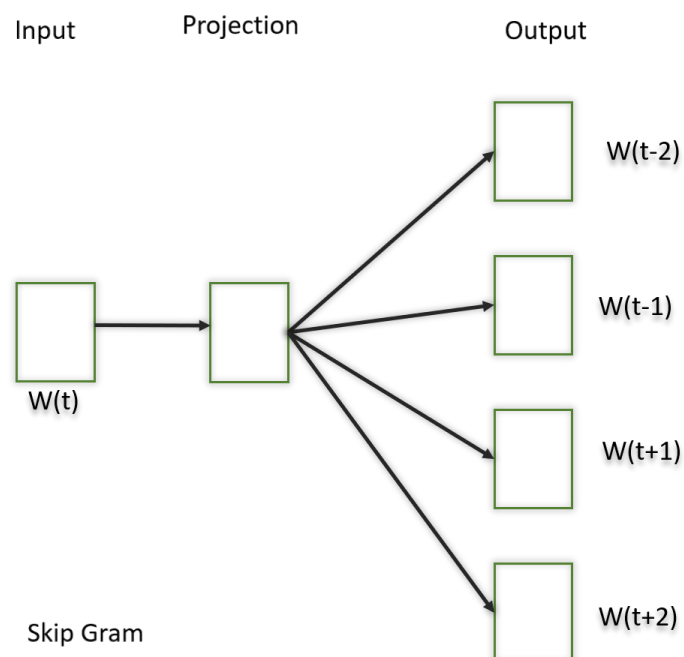
1.1) Continuous Bowl of Words (CBOW)

In this model what we do is we try to fit the neighboring words in the window to the central word.



1.2) Skip Gram

In this model, we try to make the central word closer to the neighboring words. It is the complete opposite of the CBOW model. It is shown that this method produces more meaningful embeddings.



After applying the above neural embedding methods we get trained vectors of each word after many iterations through the corpus. These trained vectors preserve syntactical or semantic information and are converted to lower dimensions. The vectors with similar meaning or semantic information are placed close to each other in space.

2) GloVe:

This is another method for creating word embeddings. In this method, we take the corpus and iterate through it and get the co-occurrence of each word with other words in the corpus. We get a co-occurrence matrix through this. The words which occur next to each other get a value of 1, if they are one word apart then 1/2, if two words apart then 1/3 and so on.

2.2.3. Word Vector Arithmetic

Word vector arithmetic is a technique for performing mathematical operations on word vectors. Word vectors are a type of numerical representation of words that are learned from a large corpus of text. They are typically represented as vectors of real numbers, where each number represents a different aspect of the meaning of the word.

The most common operations that are performed on word vectors are addition, subtraction, and multiplication. Addition and subtraction are used to find words that are semantically related to each other. For example, the word vector for "king" can be added to the word vector for "man" to get the word vector for "queen". This is because "king" and "queen" are semantically related, as they both refer to the highest-ranking member of a royal family.

$$\begin{aligned}
 V_{\text{king}} - V_{\text{man}} + V_{\text{woman}} &= V_{\text{queen}} \\
 V_{\text{bezos}} - V_{\text{amazon}} + V_{\text{tesla}} &= V_{\text{musk}} \\
 V_{\text{windows}} - V_{\text{microsoft}} + V_{\text{google}} &= V_{\text{android}}
 \end{aligned}$$

Figure : Examples of word-vector arithmetic

Multiplication is used to find words that are similar in meaning, but not necessarily related in the same way. For example, the word vector for "king" can be multiplied by the word vector for "woman" to get the word vector for "empress". This is because "king" and "empress" are both associated with power and authority, but they do not have the same gender associations.

Word vector arithmetic can be used to solve a variety of natural language processing tasks, such as

- Analogy resolution: Given a word analogy such as "king is to man as queen is to _", word vector arithmetic can be used to find the word that best completes the analogy.
- Semantic similarity: Word vector arithmetic can be used to measure the semantic similarity between two words.
- Word sense disambiguation: Word vector arithmetic can be used to disambiguate the meaning of a word in a particular context.

Word vector arithmetic is a powerful tool for natural language processing, but it is important to note that it is not a perfect solution. The results of word vector arithmetic can be sensitive to the training data that is used to learn the word vectors. Additionally, word vectors do not capture all of the nuances of human language.

Despite these limitations, word vector arithmetic is a valuable tool for natural language processing. It can be used to solve a variety of tasks, and it is constantly being improved as new research is conducted.

2.2.4. word2viz

Word2viz is a web-based tool for visualizing word embeddings. Word embeddings are a type of numerical representation of words that are learned from a large corpus of text. They are typically represented as vectors of real numbers, where each number represents a different aspect of the meaning of the word.

Word2viz allows you to visualize the relationships between words by plotting them in a two-dimensional space. The closer two words are in the plot, the more semantically similar they are. You can also use Word2viz to explore the relationships between words in a specific context.

To use Word2viz, you first need to choose a set of word embeddings. Word2viz supports a variety of word embedding models, including GloVe, Word2Vec, and FastText. Once you have chosen a word embedding model, you can upload the word embeddings to Word2viz.

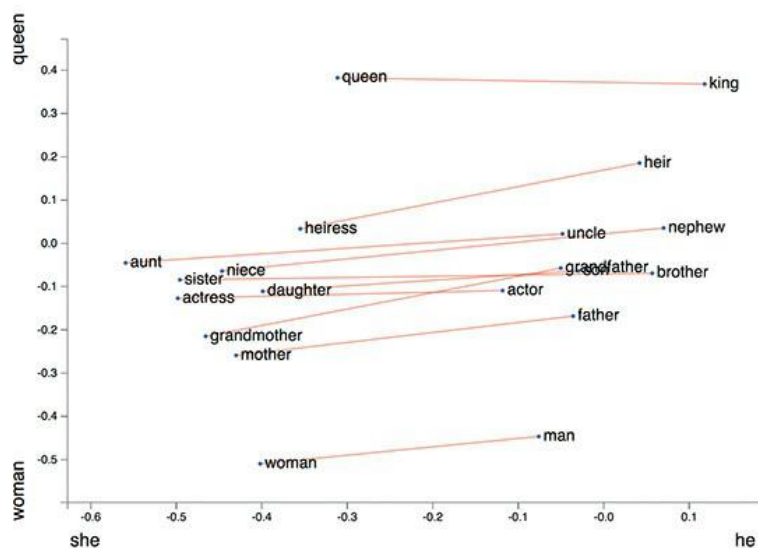
Word2viz also allows you to specify a context for the word embeddings. This can be useful for exploring the relationships between words in a specific domain or topic. For example, you could specify the context of "technology" to explore the relationships between words related to technology.

Word2viz is a powerful tool for visualizing word embeddings. It can be used to explore the relationships between words, to identify semantically similar words, and to understand the meaning of words in a specific context.

Here are some of the features of Word2viz:

- Supports a variety of word embedding models
- Allows you to specify a context for the word embeddings
- Provides interactive visualizations of the word embeddings
- Allows you to explore the relationships between words
- Identifies semantically similar words
- Understands the meaning of words in a specific context

Word2viz is a free and open-source tool. You can find more information about Word2viz on the project's website: <https://lamiowce.github.io/word2viz/>



Explore word analogies

What do you want to see?

Gender analogies

Modify words

Type a new word...

Add

Type a new word...

Type a new word...

Add pair

X axis:

she

he

Y axis:

woman

queen

Change axes labels

Interactive visualization of word analogies in GloVe. *Hover* to highlight, *double-click* to remove. *Change axes* by specifying word differences, on which you want to project. Uses (compressed) pre-trained word vectors from [glove.6B.50d](#). Made by Julia Bazińska under the mentorship of Piotr Migdał (2017).

2.2.5. Localist Versus Distributed Representations

Localist and distributed representations are two different ways of representing information in the brain.

- **Localist representations** are a type of representation in which each item is represented by a single unit. For example, the word "dog" might be represented by a single neuron. This type of representation is often used for things that are discrete and well-defined, such as numbers or letters.
- **Distributed representations** are a type of representation in which each item is represented by a pattern of activation across a set of units. For example, the word "dog" might be represented by a pattern of activation across a set of neurons, where each neuron represents a different aspect of the meaning of the word, such as its sound, its meaning, or its visual appearance. This type of representation is often used for things that are continuous or ambiguous, such as words or concepts.

There are advantages and disadvantages to both localist and distributed representations.

- **Localist representations** are often easier to learn and remember. This is because each item has its own dedicated unit, so it is easier to associate the item with its representation. However, localist representations can be inefficient, as they require a lot of units to represent a large number of items.
- **Distributed representations** are more efficient, as they can represent a large number of items with a smaller number of units. However, distributed representations can be more difficult to learn and remember, as the relationship between the item and its representation is more complex.

In general, localist representations are better for tasks that require precise identification of individual items, such as object recognition. Distributed representations are better for tasks that require understanding the relationships between items, such as language understanding.

In the context of natural language processing, localist representations are often used for tasks such as part-of-speech tagging and named entity recognition. Distributed representations are often used for tasks such as word sense disambiguation and machine translation.

The choice of whether to use localist or distributed representations depends on the specific task at hand. There is no single "best" representation, and the best choice will vary depending on the data and the task.

2.3. Elements of Natural Human Language

Natural Language Processing (NLP) is a field of study that focuses on the interaction between computers and human language. It encompasses various techniques and methodologies to enable computers to understand, interpret, and generate human language. There are several essential elements that form the foundation of Natural Language Processing. Let's explore them:

Representation	Traditional ML	Deep Learning	Audio-Only
Phonology	All phonemes	Vectors	True
Morphology	All morphemes	Vectors	False
Words	One-hot encoding	Vectors	False
Syntax	Phrase rules	Vectors	False
Semantics	Lambda calculus	Vectors	False

Table: Traditional machine learning and deep learning representations, by natural language element

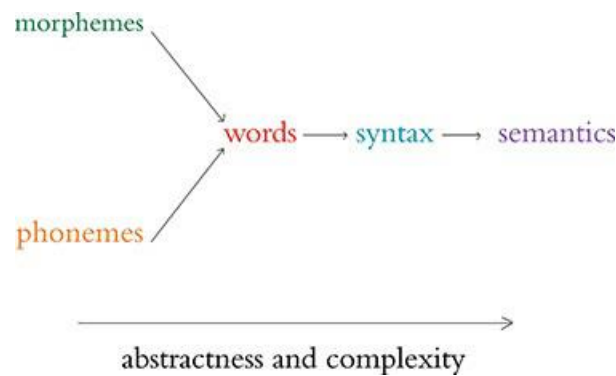
Phonology is the study of how sounds are used to make words in a language. Every language has a different set of sounds, and these sounds are combined to form words. The traditional way to study phonology is to break down speech into small pieces and label each piece with a specific sound. With deep learning, we can train a model to automatically learn the sounds of a language from speech data. This model can then be used to represent words as combinations of sounds.

Morphology is concerned with the forms of words. Like phonemes, every language has a specific set of *morphemes*, which are the smallest units of language that contain some meaning.

For example, the three morphemes out, go, and ing combine to form the word outgoing. The traditional ML approach is to identify morphemes in text from a list of all the morphemes in a given language. With deep learning, we train a model to predict the occurrence of particular morphemes.

Hierarchically deeper layers of artificial neurons can then combine multiple vectors (e.g., the three representing out, go, and ing) into a single vector representing a word.

Phonemes (when considering audio) and morphemes (when considering text) combine to form words. Whenever we work with natural language data, we work at the word level.



Words are combined to generate **syntax**. Syntax and Essentially, tokenization is the use of characters like commas, periods, and whitespace to assume where one word ends and the next begins.

morphology together constitute the entirety of a language's grammar.

Syntax is the arrangement of words into phrases and phrases into sentences in order to convey meaning in a way that is consistent across the users of a given language.

In the traditional ML approach, phrases are bucketed into discrete, formal linguistic categories. With deep learning, we employ vectors (surprise, surprise!). Every word and every phrase in a section of text can be represented by a vector in n-dimensional space, with layers of artificial neurons combining words into phrases.

Semantics is concerned with the meaning of words, phrases, and sentences. It focuses on understanding the context and intent behind the language used. This element of NLP involves techniques like word sense disambiguation, named entity recognition, and sentiment analysis. Semantic understanding is essential for tasks such as question answering, text summarization, and sentiment analysis

2.4. Google Duplex

Google Duplex is a research project by Google AI that is developing a technology that can have natural conversations with humans over the phone. The goal of Google Duplex is to create a more natural and human-like experience for users when interacting with voice assistants.

Google Duplex is still under development, but it has already made some impressive progress. In a recent demonstration, Google Duplex was able to successfully make a restaurant reservation over the phone. The caller sounded so natural that the restaurant staff did not realize they were talking to a machine.

Google Duplex uses a variety of techniques to achieve its naturalness. These techniques include:

- **Natural language processing:** Google Duplex uses natural language processing to understand the meaning of the user's words.

- **Speech synthesis:** Google Duplex uses speech synthesis to generate natural-sounding speech.
- **Dialogue management:** Google Duplex uses dialogue management to keep the conversation on track and to handle unexpected situations.

Google Duplex is a promising technology that has the potential to revolutionize the way we interact with voice assistants. However, there are still some challenges that need to be addressed before Google Duplex can be widely released. These challenges include:

- **Accuracy:** Google Duplex needs to be more accurate in understanding the user's words and generating natural-sounding speech.
- **Security:** Google Duplex needs to be secure so that users can be confident that their conversations are private.
- **Acceptance:** Google Duplex needs to be accepted by users before it can be widely used.

Chapter 3: Artificial Neural Networks

- The Input Layer
- Dense Layers
- A Hot Dog-Detecting Dense Network
 - Forward Propagation through the First Hidden Layer
 - Forward Propagation through Subsequent Layers
- The Softmax Layer of a Fast Food-Classifying Network (softmax_demo.ipynb)
- Revisiting our Shallow Neural Network

3.1.THE INPUT LAYER

In our *Shallow Net in Keras*, we crafted an artificial neural network with the following layers:

1. An *input* layer consisting of 784 neurons, one for each of the 784 pixels in an MNIST image
2. A *hidden* layer composed of 64 sigmoid neurons
3. An *output* layer consisting of 10 softmax neurons, one for each of the 10 classes of digits

Of these three, the input layer is the most straightforward to detail. We start with it and then move on to discussion of the hidden and output layers.

Neurons in the input layer don't perform any calculations; they are simply placeholders for input data. This placeholder is essential because the use of artificial neural networks involves performing computations on matrices that have predefined dimensions. At least one of these predefined dimensions in the network architecture corresponds directly to the shape of the input data.

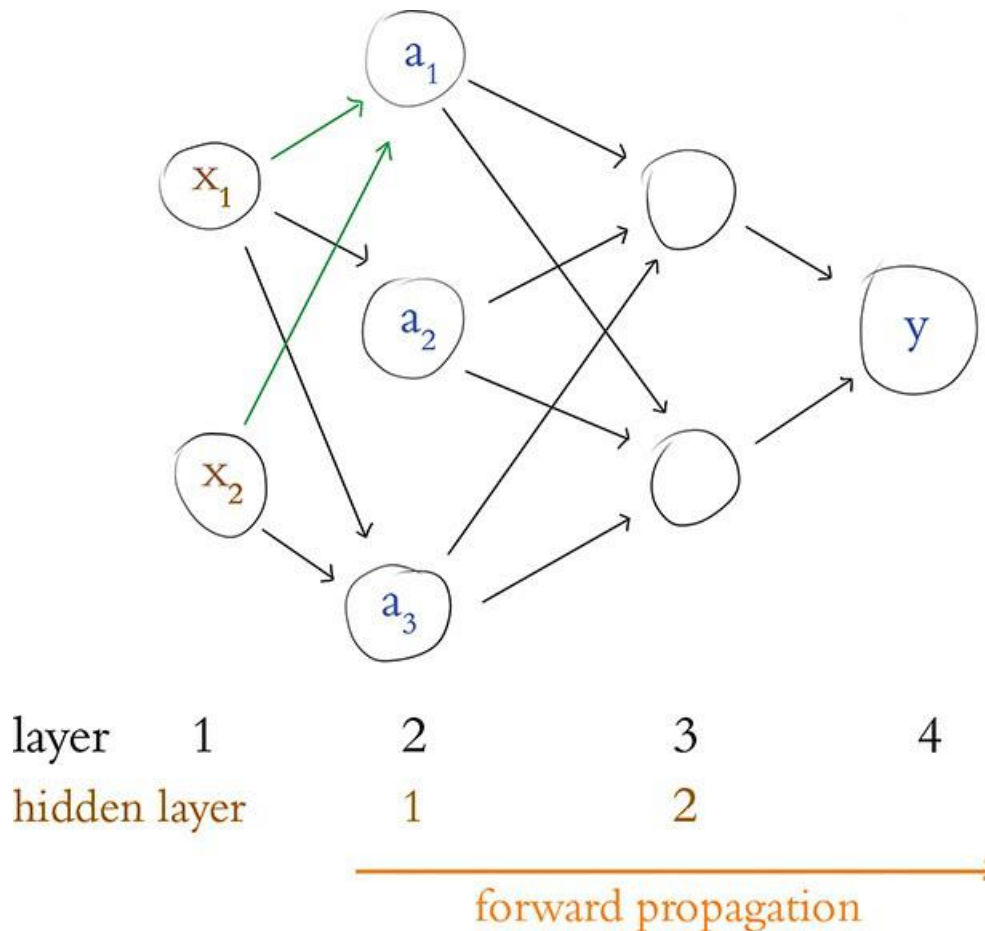
3.2.DENSE LAYERS

There are many kinds of hidden layers, the most general type is the dense layer, which can also be called a fully connected layer.

Dense layers are found in many deep learning architectures. Their definition is uncomplicated: Each of the neurons in a given dense layer receive information from every one of the neurons in the preceding layer of the network. In other words, a dense layer is fully connected to the layer before it!

3.3. A HOT DOG-DETECTING DENSE NETWORK

A hot dog-detecting dense network is a type of artificial neural network that can be used to identify hot dogs in images. The network would typically have an input layer, one or more hidden layers, and an output layer. The input layer would receive the image data. The hidden layers would process the data and extract features that are relevant to hot dogs. The output layer would then classify the image as either a hot dog or not a hot dog.



The first input neuron, x_1 , represents the volume of ketchup (in, say, milliliters, which abbreviates to mL) on the object being considered by the network. (We are no longer working with perceptrons, so we are no longer restricted to binary inputs only.)

The second input neuron, x_2 , represents milliliters of mustard.

We have two dense hidden layers.

- The first hidden layer has three ReLU neurons.
- The second hidden layer has two ReLU neurons.

The output neuron is denoted by y in the network. This is a binary classification problem, so, this neuron should be sigmoid. As in our perceptron examples in, $y = 1$ corresponds to the presence of a hot dog and $y = 0$ corresponds to the presence of some other object.

Forward Propagation Through the First Hidden Layer

Having described the architecture of our hot dog-detecting network, let's turn our attention to its functionality by focusing on the neuron labeled a_1 .

This particular neuron, like its siblings a_2 and a_3 , receives input regarding a given object's "ketchup-y-ness" and "mustard-y-ness" from x_1 and x_2 , respectively.

Despite receiving the same data as **a2** and **a3** , **a1** treat these data uniquely by having its own unique parameters.

To grasp this behavior, we use the following equation

$$z = w \cdot x + b$$

For neuron a1

- we consider that it has two inputs from the preceding layer: **x1** and **x2** .
- This neuron also has two weights: **w1** (which applies to the importance of the ketchup measurement **x1**) and **w2** (which applies to the importance of the mustard measurement **x2**).
- With these five pieces of information, we can calculate **z**, the weighted input to that neuron:

$$z = w \cdot x + b$$

$$z = (w1x1 + w2x2) + b$$

In turn, with the **z** value for the neuron labeled **a1** , we can calculate the activation **a1** it outputs. Because the neuron labeled **a1** is a ReLU neuron

$$a = \max(0, z)$$

To make this computation of the output of neuron **a** tangible, let's concoct some numbers and work through the arithmetic together:

x1 is 4.0 mL of ketchup for a given object presented to the network

x2 is 3.0 mL of mustard for that same object

$$w1 = -0.5$$

$$w2 = 1.5$$

$$b = -0.9$$

To calculate **z**

$$z = w \cdot x + b$$

$$= w1x1 + w2x2 + b$$

$$= -0.5 \times 4.0 + 1.5 \times 3.0 - 0.9$$

$$= -2 + 4.5 - 0.9$$

$$= 1.6$$

Finally, to compute **a1**—the activation output of the neuron labeled **a1**:

$$a = \max(0, z)$$

$$= \max(0, 1.6)$$

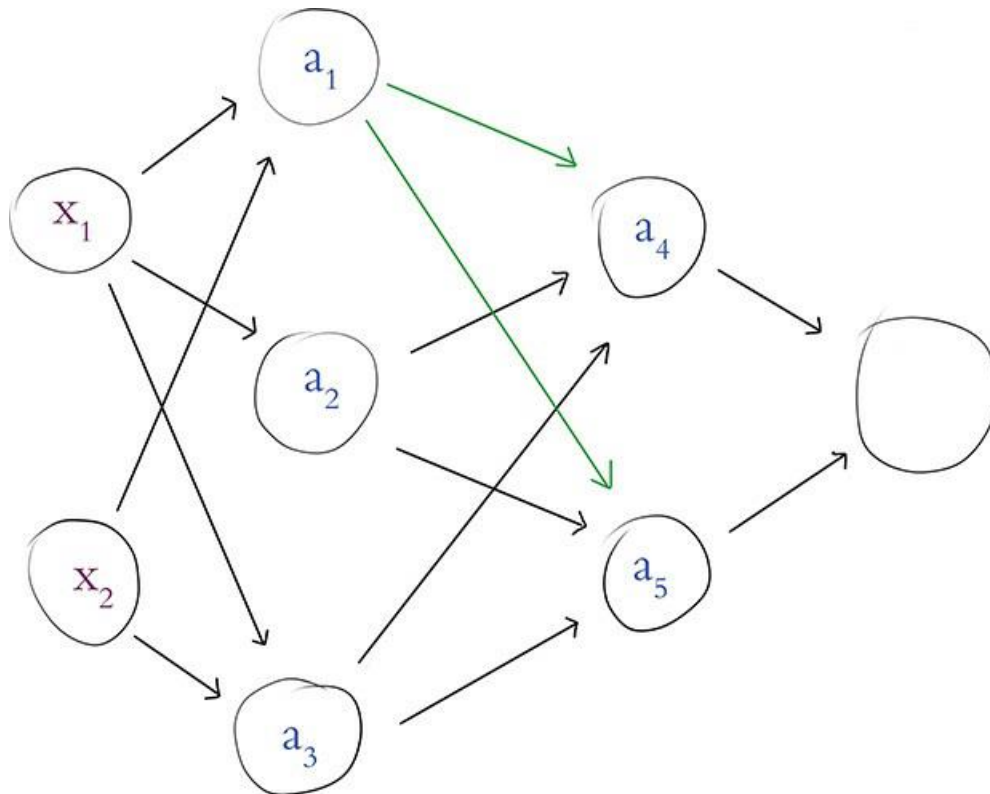
$$= 1.6$$

executing the calculations through an artificial neural network from the input layer (the **x** values) through to the output layer (**y**) is called *forward propagation*

To forward propagate through the remaining neurons of the first hidden layer—that is, to calculate the a values for the neurons labeled a_2 and a_3 —we would follow the same process as we did for the neuron labeled a_1 .

Forward Propagation Through Subsequent Layers

The process of forward propagating through the remaining layers of the network is essentially the same as propagating through the first hidden layer, but for clarity's sake, let's work through an example together.



In above Figure, we assume that we've already calculated the activation value a for each of the neurons in the first hidden layer.

Returning our focus to the neuron labeled a_1 , the activation it outputs ($a_1 = 1.6$) becomes one of the three inputs into the neuron labeled a_4 (and, as highlighted in the figure, this same activation of $a = 1.6$ is also fed as one of the three inputs into the neuron labeled a_5).

To provide an example of forward propagation through the second hidden layer, let's compute a for the neuron labeled a_4 .

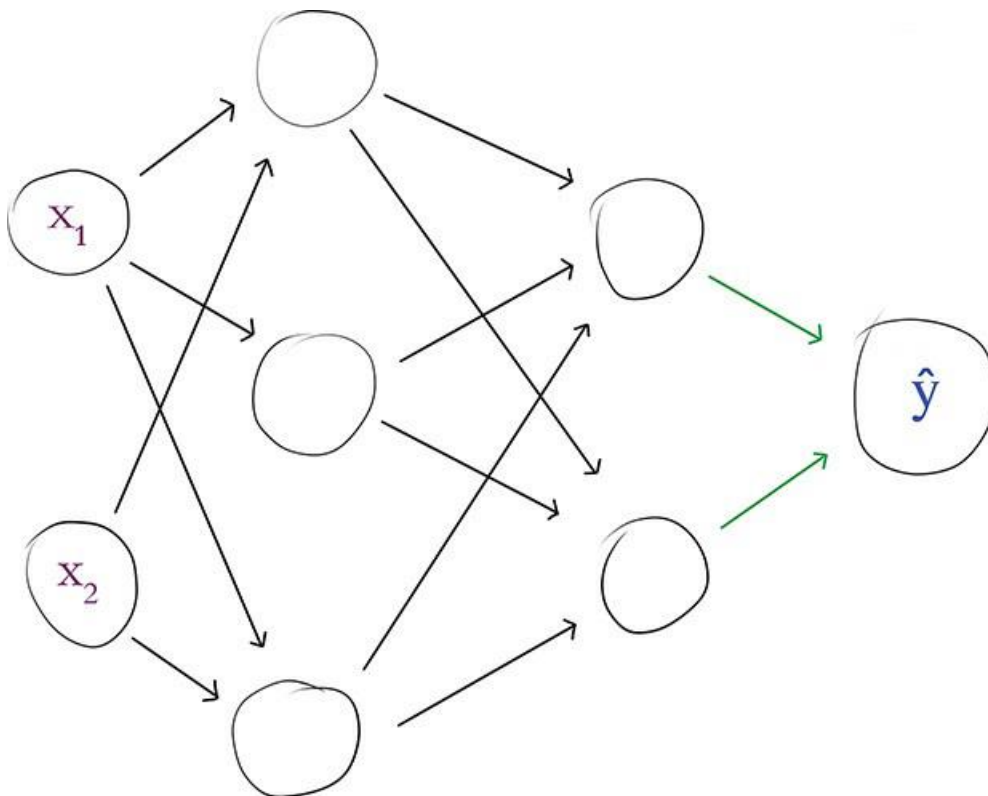
$$\begin{aligned}
 a &= \max(0, z) \\
 &= \max(0, (w \cdot x + b)) \\
 &= \max(0, (w_1x_1 + w_2x_2 + w_3x_3 + b))
 \end{aligned}$$

As we propagate through the second hidden layer, the only twist is that the layer's inputs (i.e., x in the equation $w \cdot x + b$) do not come from outside the network; instead they are provided by the first hidden layer. Thus,

- x_1 is the value $a = 1.6$, which we obtained earlier from the neuron labeled a_1

- x_2 is the activation output a (whatever it happens to equal) from the neuron labeled a_2
- x_3 is likewise a unique activation a from the neuron labeled a_3
- The unique parameters w_1 , w_2 , w_3 , and b for this neuron would lead it to output a unique activation of its own.

$$\begin{aligned}
 z &= w \cdot x + b \\
 &= w_1 x_1 + w_2 x_2 + b \\
 &= 1.0 \times 2.5 + 0.5 \times 2.0 - 5.5 \\
 &= 3.5 - 5.5 \\
 &= -2.0
 \end{aligned}$$



The output neuron is sigmoid, so to compute its activation a we pass its z value through the sigmoid function

$$\begin{aligned}
 a &= \sigma(z) \\
 &= \frac{1}{1 + e^{-z}} \\
 &= \frac{1}{1 + e^{-(-2.0)}} \\
 &\approx 0.1192
 \end{aligned}$$

3.4. THE SOFTMAX LAYER OF A FAST FOODCLASSIFYING NETWORK

The sigmoid neuron suits us well as an output neuron if we're building a network to distinguish two classes, such as a blue dot versus an orange dot, or a hot dog versus something other than a hot dog.

In many other circumstances, however, you have more than two classes to distinguish between. For example, the MNIST dataset consists of the 10 numerical digits, so our Shallow Net in Keras had to accommodate 10 output probabilities—one representing each digit.

When concerned with a multiclass problem, the solution is to use a softmax layer as the output layer of our network. Softmax is in fact the activation function that we specified for the output layer in our Shallow Net in Keras.

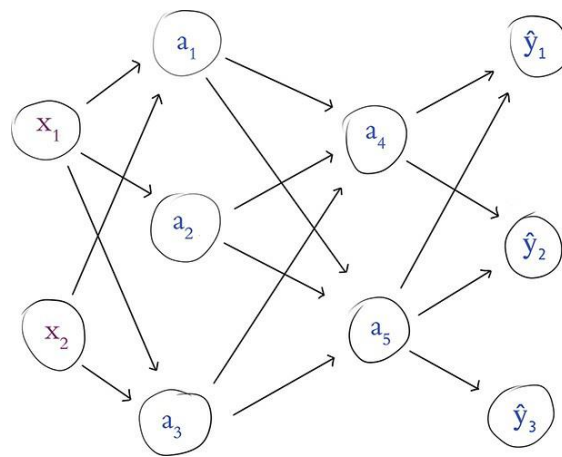


Figure: Our food-detecting network, now with three softmax neurons in the output layer

except that instead of having a single output neuron, we now have three. This multiclass output layer is still dense, so each of the three neurons receives information from both of the neurons in the final hidden layer. Continuing on with our proclivity for fast food, let's say that now:

y1 represents hot dogs.

y2 is for burgers.

y3 is for pizza

Based on the information that the three neurons receive from the final hidden layer, they individually use the formula $\mathbf{w} \cdot \mathbf{x} + \mathbf{b}$ to calculate three unique (and, for the purposes of this example, contrived) z values:

- **z** for the neuron labeled \hat{y}_1 , which represents hot dogs, comes out to -1.0.
- For the neuron labeled \hat{y}_2 , which represents burgers, **z** is 1.0.
- For the pizza neuron \hat{y}_3 , **z** comes out to 5.0.

After importing our dependency, we create a list named z to store our three z values:

`z = [-1.0, 1.0, 5.0]`

Applying the softmax function to this list involves a three-step process.

1. The first step is to calculate the exponential of each of the z values.

$\exp(z[0])$ comes out to 0.3679 for hot dog.

$\exp(z[1])$ gives us 2.718 for burger.

$\exp(z[2])$ gives us the much, much larger (exponentially so!) 148.4 for pizza.

2. The second step of the softmax function is to sum up our exponentials:

$\text{total} = \exp(z[0]) + \exp(z[1]) + \exp(z[2])$

3. With this total variable we can execute the third and final step, which provides proportions for each of our three classes relative to the sum of all of the classes:

- $\exp(z[0]) / \text{total}$ outputs a \hat{y}_1 value of 0.002428, indicating that the network estimates there's a ~0.2 percent chance that the object presented to it is a hot dog.
- $\exp(z[1]) / \text{total}$ outputs a \hat{y}_2 value of 0.01794, indicating an estimated ~1.8 percent chance that it's a burger.
- $\exp(z[2]) / \text{total}$ outputs a \hat{y}_3 value of 0.9796, for an estimated ~98.0 percent chance that the object is pizza.

3.4. REVISITING OUR SHALLOW NETWORK

the three lines of Keras code we use to architect a shallow neural network for classifying MNIST digits are

```
model = Sequential()  
model.add(Dense(64, activation='sigmoid', input_shape=(784,)))  
model.add(Dense(10, activation='softmax'))
```

over these three lines of code, we instantiate a model object and add layers of artificial neurons to it. By calling the `summary()` method on the model, we see the model-summarizing table provided below

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 64)	50240
dense_2 (Dense)	(None, 10)	650
Total params: 50,890		
Trainable params: 50,890		
Non-trainable params: 0		

The input layer performs no calculations and never has any of its own parameters, so no information on it is displayed directly.

The first row in the table, therefore, corresponds to the first hidden layer of the network. The table indicates that this layer:

Is called `dense_1`; this is a default name because we did not designate one explicitly

Is a Dense layer

Is composed of 64 neurons

Has 50,240 parameters associated with it, broken down into the following:

50,176 weights, corresponding to each of the 64 neurons in this dense layer receiving input from each of the 784 neurons in the input layer (64×784)

Plus 64 biases, one for each of the neurons in the layer

Giving us a total of 50,240 $n_{parameters} = n_w + n_b$

$$= 50176 + 64 = 50240$$

The second row of the table corresponds to the model's output layer. The table tells us that this layer:

Is called dense_2

Is a Dense layer, as we specified it to be

Consists of 10 neurons—again, as we specified Has 650 parameters associated with it, as follows:

640 weights, corresponding to each of the 10 neurons receiving input from each of the 64 neurons in the hidden layer (64×10)

Plus 10 biases, one for each of the output neurons

From the parameter counts for each layer, we can calculate for ourselves the Total params line displayed

$$n_{total} = n1 + n2$$

$$= 50240 + 650$$

$$= 50890$$

All 50,890 of these parameters are Trainable params because—during the subsequent `model.fit()` call in the Shallow Net in Keras they are permitted to be tuned during model training.

Chapter 4: Training Deep Networks

4.1. Cost Functions

- Quadratic Cost
- Saturated Neurons
- Cross-Entropy Cost

The main goal of any neural network is to make accurate predictions. A cost function **helps to quantify how far the neural network's predictions are from the actual values**. It is a measure of the error between the predicted output and the actual output.

Types of Cost Functions

There are different types of cost functions, and the choice of cost function depends on the type of problem being solved. Here are some commonly used cost functions:

1. Mean Squared Error (MSE) / Quadratic Cost

The mean squared error is one of the most popular cost functions for regression problems. It measures the average squared difference between the predicted and actual values. The formula for MSE is:

$$\text{MSE} = (1/n) * \sum (y - \hat{y})^2$$

Where:

- n is the number of samples in the dataset
- y is the actual value
- \hat{y} is the predicted value

2. Binary Cross-Entropy

The binary cross-entropy cost function is used for binary classification problems. It measures the difference between the predicted and actual values in terms of probabilities. The formula for binary cross-entropy is:

$$\text{Binary Cross-Entropy} = - (1/n) * \sum (y * \log(\hat{y}) + (1 - y) * \log(1 - \hat{y}))$$

Where:

- n is the number of samples in the dataset
- y is the actual value (0 or 1)
- \hat{y} is the predicted probability (between 0 and 1)

3. Categorical Cross-Entropy

The categorical cross-entropy cost function is used for multi-class classification problems. It measures the difference between the predicted and actual values in terms of probabilities. The formula for categorical cross-entropy is:

$$\text{Categorical Cross-Entropy} = - (1/n) * \sum \sum (y(i,j) * \log(\hat{y}(i,j)))$$

Where:

- n is the number of samples in the dataset
- $y(i,j)$ is the actual value of the i-th sample for the j-th class
- $\hat{y}(i,j)$ is the predicted probability of the i-th sample for the j-th class

How to Choose a Cost Function

Choosing the right cost function is crucial for the performance of a neural network. Here are some factors to consider when choosing a cost function:

a) Type of Problem

The type of problem being solved determines the type of cost function to use. For example, regression problems require a different cost function than classification problems.

b) Output Activation Function

The output activation function can also influence the choice of cost function. For example, if the output activation function is sigmoid, then the binary cross-entropy cost function is a good choice. If the output activation function is softmax, then the categorical cross-entropy cost function is a good choice.

c) Network Architecture

The network architecture can also influence the choice of cost function. For example, if the network has multiple outputs, then the multi-task loss function is a good choice.

Saturated Neurons

In neural networks, a saturated neuron is a neuron whose output has reached its maximum or minimum value. This can happen when the weights of the neuron are too large or too small. Saturated neurons can prevent the network from learning, so it is important to avoid them.

There are two main types of saturated neurons:

- **Linear neurons:** Linear neurons are neurons whose output is a linear function of their inputs. This means that the output of a linear neuron will always be between 0 and 1, regardless of the values of its inputs. Linear neurons can become saturated when the weights of the neuron are too large or too small.
- **Sigmoid neurons:** Sigmoid neurons are neurons whose output is a sigmoid function of their inputs. Sigmoid functions are S-shaped curves that output values between 0 and 1. Sigmoid neurons can become saturated when the weights of the neuron are too large or too small.

Saturated neurons can prevent the network from learning because they do not contribute to the error signal. The error signal is the difference between the predicted output of the network and the desired output. When a neuron is saturated, its output is always the same, regardless of the input. This means that the neuron does not contribute to the error signal, and the network cannot learn from it.

There are a few things that can be done to avoid saturated neurons:

- **Use a different activation function:** Sigmoid neurons are more likely to become saturated than linear neurons. If you are using a sigmoid activation function, you can try using a different activation function, such as the ReLU activation function.
- **Use a smaller learning rate:** A smaller learning rate will help the network to avoid overshooting the weights and becoming saturated.
- **Regularize the network:** Regularization is a technique that helps to prevent overfitting. There are a variety of regularization techniques, such as L1 regularization and L2 regularization.

4.2. Optimization: Learning to Minimize Cost

- Gradient Descent
- Learning Rate
- Batch Size and Stochastic Gradient Descent
- Escaping the Local Minimum

Cost functions provide us with a quantification of how incorrect our model's estimate of the ideal y is. This is most helpful because it arms us with a metric we can leverage to reduce our network's incorrectness.

The primary approach for minimizing cost in deep learning paradigms is to pair an approach called gradient descent with another one called backpropagation.

These approaches are *optimizers* and they enable the network to *learn*. This learning is accomplished by adjusting the model's parameters so that its estimated \hat{y} gradually converges toward the target of y , and thus the cost decreases.

4.2.1.Gradient Descent

Gradient descent is a handy, efficient tool for adjusting a model's parameters with the aim of minimizing cost, particularly if you have a lot of training data available. It is widely used across the field of machine learning, not only in deep learning.

Gradient descent is an optimization algorithm commonly used in machine learning to train models. It works by iteratively updating the model's parameters in the direction of the negative gradient of the loss function. This means that gradient descent takes steps towards the local minimum of the loss function, which is the point where the model makes the fewest mistakes.

Gradient descent is a powerful algorithm, but it is important to note that it does not guarantee finding the global minimum of the loss function. It is also important to choose a good learning rate, which determines how large the steps taken by gradient descent are. If the learning rate is too small, gradient descent will converge slowly, but if it is too large, gradient descent may overshoot the minimum and diverge.

Here is a simple example of how gradient descent works:

1. Start with a random set of model parameters.
2. Calculate the loss function for the current model parameters.
3. Calculate the gradient of the loss function with respect to the model parameters.
4. Update the model parameters in the direction of the negative gradient, using a learning rate.
5. Repeat steps 2-4 until the loss function converges to a minimum.

Types Gradient descent

There are three main types of gradient descent:

- **Batch gradient descent:** Batch gradient descent calculates the gradient of the loss function over the entire training dataset before updating the model parameters. This is the slowest type of gradient descent, but it is also the most accurate.
- **Stochastic gradient descent (SGD):** SGD calculates the gradient of the loss function over a single training example before updating the model parameters. This is the fastest type of gradient descent, but it is also the least accurate.

- **Mini-batch gradient descent:** Mini-batch gradient descent calculates the gradient of the loss function over a small batch of training examples before updating the model parameters. This is a compromise between batch gradient descent and SGD, and it is often the best choice for practical applications.

In addition to these three main types of gradient descent, there are many other variants of gradient descent that have been developed to improve its performance. Some of these variants include:

- **Momentum:** Momentum adds a term to the gradient update that is proportional to the previous gradient update. This helps the algorithm to converge more quickly to the minimum.
- **Nesterov momentum:** Nesterov momentum is a variant of momentum that calculates the gradient at the next iteration before updating the model parameters. This can further improve the convergence rate of the algorithm.
- **Adam:** Adam is a variant of gradient descent that uses an adaptive learning rate. This means that the learning rate is adjusted dynamically during the training process.

The best type of gradient descent to use for a particular problem will depend on the specific characteristics of the problem. For example, if the problem is very noisy, SGD may not be a good choice because it is sensitive to noise. If the problem is very large, batch gradient descent may not be a good choice because it is too slow.

4.2.2 Learning Rate

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. It is one of the most important hyperparameters when configuring your neural network.

A good learning rate will allow the model to converge to the minimum of the loss function in a reasonable amount of time without overshooting or diverging.

There are a few different ways to choose a learning rate

- Start with a small learning rate and increase it gradually until the model starts to overshoot or diverge.
- Use a heuristic, such as the rule of thumb that the learning rate should be equal to the inverse of the square root of the number of parameters in the model.
- Try using an adaptive learning rate algorithm.
- If the model is overfitting, try reducing the learning rate.
- If the model is not converging, try increasing the learning rate.

4.2.3. Batch Size and Stochastic Gradient Descent

The *stochastic* variant of gradient descent. With this variation, we split our training data into *mini-batches*—small subsets of our full training dataset—to render gradient descent both manageable and productive

We can set stochastic gradient descent by setting our optimizer to SGD in the `model.compile()` step.

Further, in the subsequent line of code when we called the `model.fit()` method, we set `batch_size` to 128 to specify the size of our mini-batches—the number of training data points that we use for a given iteration of SGD.

Like the learning rate η presented earlier in this chapter, *batch size* is also a model hyperparameter.

Before carrying out any training, we initialize our network with random values for each neuron's parameters **w** and **b**. To begin the first epoch of training:

1. We shuffle and divide the training images into mini-batches of 128 images each.

2. By forward propagation, information about the 128 images is processed by the network, layer through layer, until the output layer ultimately produces \hat{y} values.
3. A cost function (e.g., cross-entropy cost) evaluates the network's \hat{y} values against the true y values, providing a cost C for this particular mini-batch of 128 images
4. To minimize cost and thereby improve the network's estimates of y given x , the gradient descent part of stochastic gradient descent is performed: Every single w and b parameter in the network is adjusted proportional to how much each contributed to the error (i.e., the cost) in this batch (note that the adjustments are scaled by the learning rate hyperparameter η).

These four steps constitute a *round of training*, as summarized by below image

Round of Training:

1. Sample a mini-batch of x values
2. Forward propagate x through network to estimate y with \hat{y}
3. Calculate cost C by comparing y and \hat{y}
4. Descend gradient of C to adjust w and b , enabling x to better predict y




Figure 8.6 captures how rounds of training are repeated until we run out of training images to sample.

The sampling in step 1 is done *without replacement*, meaning that at the end of an epoch each image has been seen by the algorithm only once, and yet between different epochs the mini-batches are sampled randomly. After a total of 468 rounds, the final batch contains only 96 samples.

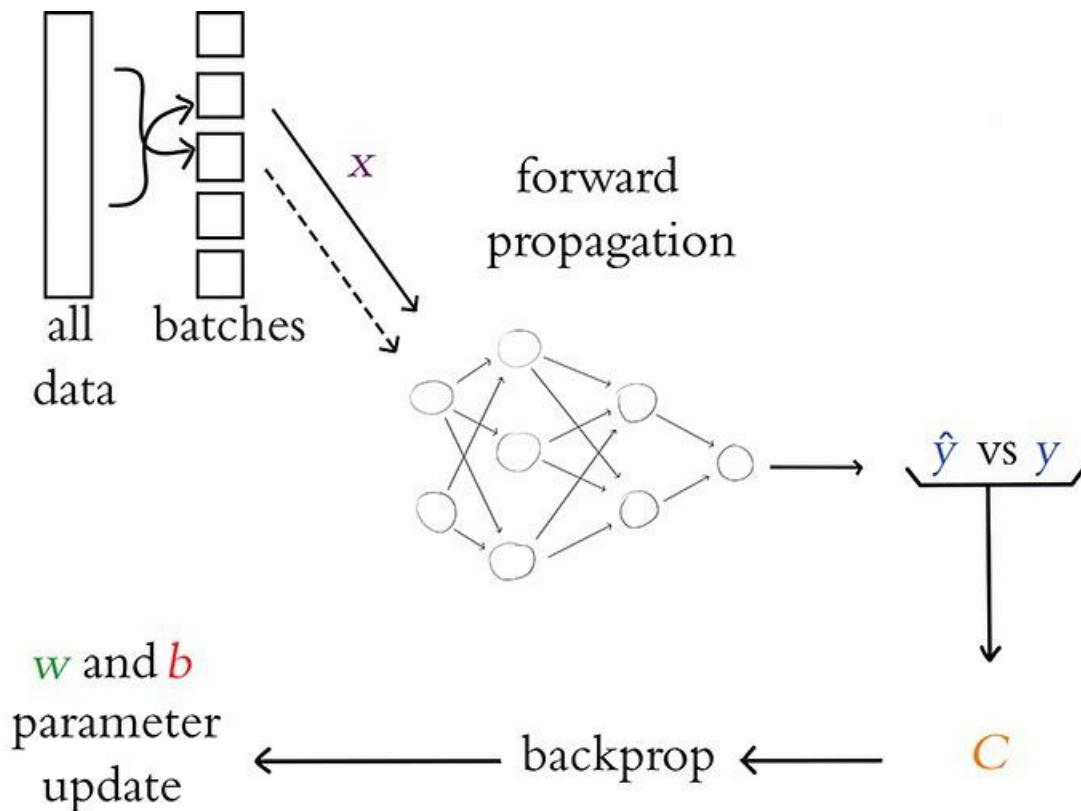


Figure 8.6 An outline of the overall process for training a neural network with stochastic gradient descent.

- ☞ This marks the end of the first epoch of training.
- ☞ Assuming we've set our model up to train for further epochs, we begin the next epoch by replenishing our pool with all 60,000 training images.
- ☞ As we did through the previous epoch, we then proceed through a further 469 rounds of stochastic gradient descent.
- ☞ Training continues in this way until the total desired number of epochs is reached.

The total *number of epochs* that we set our network to train for is yet another hyperparameter, by the way. This hyperparameter, though, is one of the easiest to get right:

- a) If the cost on your validation data is going down epoch over epoch, and if your final epoch attained the lowest cost yet, then you can try training for additional epochs.
- b) Once the cost on your validation data begins to creep upward, that's an indicator that your model has begun to *overfit* to your training data because you've trained for too many epochs.
- c) There are methods you can use to automatically monitor training and validation cost and stop training early if things start to go awry.

In this way, you could set the number of epochs to be arbitrarily large and know that training will continue until the validation cost stops improving—and certainly before the model begins overfitting!

4.2.4. Escaping the Local Minimum

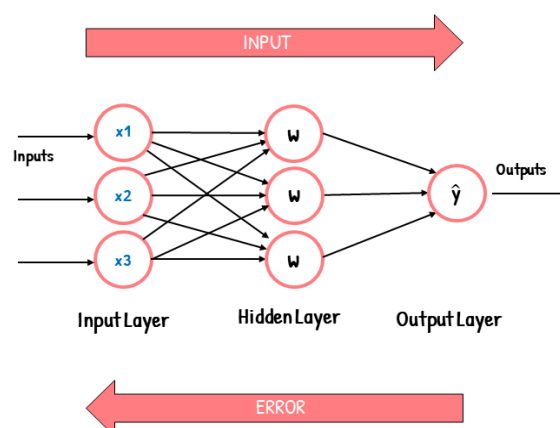
There are a number of techniques that can be used to escape the local minimum in gradient descent:

- **Use a good learning rate:** A good learning rate will help the model to escape the local minimum without overshooting or diverging.
- **Use momentum:** Momentum adds a term to the gradient update that is proportional to the previous gradient update. This helps the algorithm to escape the local minimum by preventing it from getting stuck in a narrow valley.
- **Use Nesterov momentum:** Nesterov momentum is a variant of momentum that calculates the gradient at the next iteration before updating the model parameters. This can further improve the ability of the algorithm to escape the local minimum.
- **Use an adaptive learning rate algorithm:** Adaptive learning rate algorithms automatically adjust the learning rate during training. This can help the model to escape the local minimum by allowing it to use a higher learning rate in the early stages of training and a lower learning rate in the later stages of training.
- **Use regularization:** Regularization adds a penalty to the loss function that penalizes large values of the model parameters. This can help the model to escape the local minimum by preventing it from becoming overfitting to the training data.
- **Use multiple restarts:** One way to increase the chances of escaping the local minimum is to restart the training process multiple times with different random initializations.
- **Use a large training dataset:** A larger training dataset will provide the model with more information to help it escape the local minimum.
- **Use a complex model:** A more complex model will have more parameters, which can give it more flexibility to escape the local minimum.
- **Use data augmentation:** Data augmentation can be used to create new training data from existing training data. This can help to reduce overfitting and improve the ability of the model to escape the local minimum.

4.3. BACKPROPAGATION

Backpropagation is a training algorithm used for training feedforward neural networks. It plays an important part in improving the predictions made by neural networks. This is because backpropagation is able to improve the output of the neural network iteratively.

In a feedforward neural network, the input moves forward from the input layer to the output layer. Backpropagation helps improve the neural network's output. It does this by propagating the error backward from the output layer to the input layer.



How Does Backpropagation Work?

To understand how backpropagation works, let's first understand how a feedforward network works.

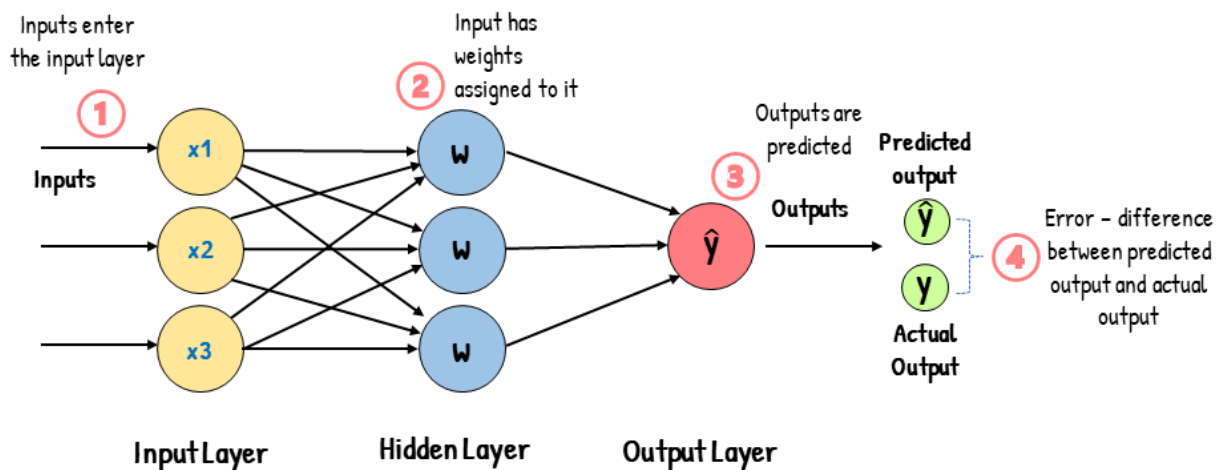
Feed Forward Networks

A feedforward network consists of an input layer, one or more hidden layers, and an output layer. The input layer receives the input into the neural network, and each input has a weight attached to it.

The weights associated with each input are numerical values. These weights are an indicator of the importance of the input in predicting the final output. For example, an input associated with a large weight will have a greater influence on the output than an input associated with a small weight.

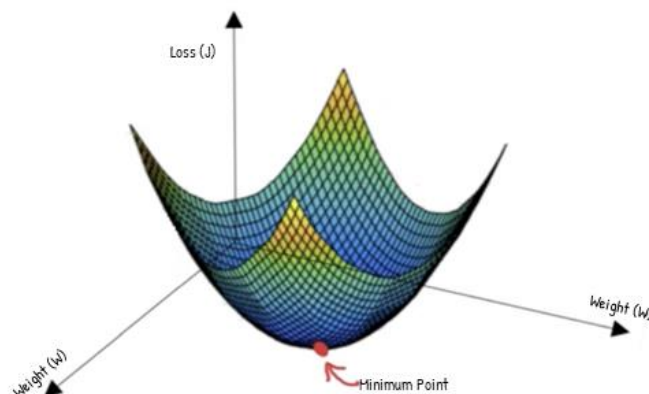
When a neural network is first trained, it is first fed with input. Since the neural network isn't trained yet, we don't know which weights to use for each input. And so, each input is randomly assigned a weight. Since the weights are randomly assigned, the neural network will likely make the wrong predictions. It will give out the incorrect output.

Feed-Forward Neural Network



When the neural network gives out the incorrect output, this leads to an output error. This error is the difference between the actual and predicted outputs. A cost function measures this error.

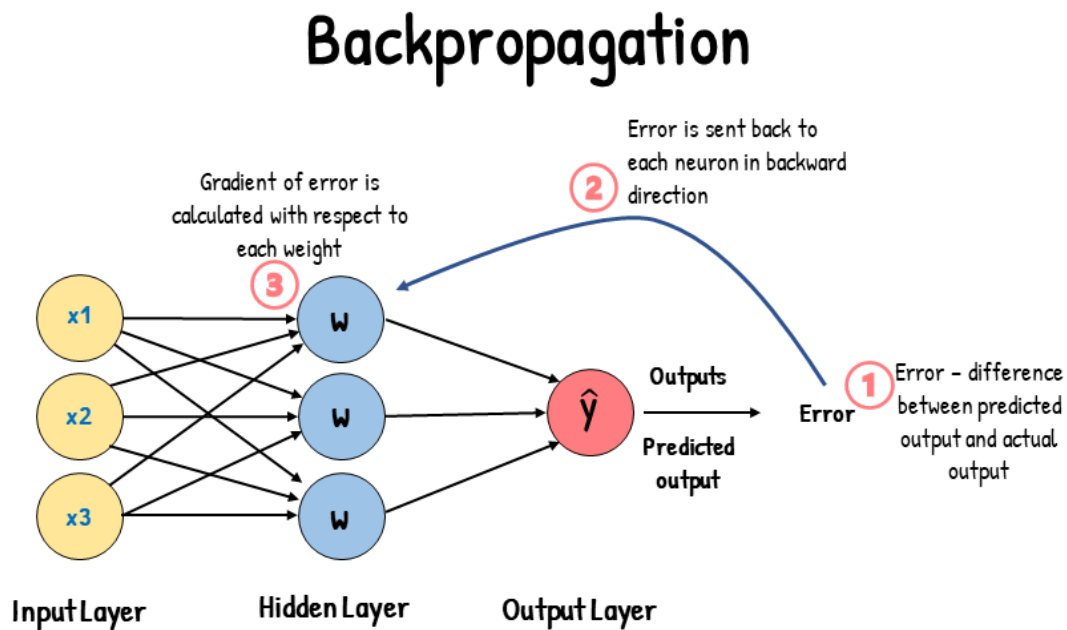
The **cost function (J)** indicates how accurately the model performs. It tells us how far-off our predicted output values are from our actual values. It is also known as the error. Because the cost function quantifies the error, we aim to minimize the cost function.



What we want is to reduce the output error. Since the weights affect the error, we will need to readjust the weights. We have to adjust the weights such that we have a combination of weights that minimizes the cost function.

This is where Backpropagation comes in...

Backpropagation allows us to readjust our weights to reduce output error. The error is propagated backward during backpropagation from the output to the input layer. This error is then used to calculate the gradient of the cost function with respect to each weight.



Essentially, backpropagation aims to calculate the negative gradient of the cost function. This negative gradient is what helps in adjusting of the weights. It gives us an idea of how we need to change the weights so that we can reduce the cost function.

Backpropagation uses the chain rule to calculate the gradient of the cost function. The chain rule involves taking the derivative. This involves calculating the partial derivative of each parameter. These derivatives are calculated by differentiating one weight and treating the other(s) as a constant. As a result of doing this, we will have a gradient.

Since we have calculated the gradients, we will be able to adjust the weights.

Backpropagation vs. Gradient Descent

	Backpropagation	Gradient Descent
Definition	An algorithm for calculating the gradients of the cost function	Optimization algorithm used to find the weights that minimize the cost function
Requirements	Differentiation via the chain rule	<ul style="list-style-type: none"> •Gradient via Backpropagation •Learning rate
Process	Propagating the error backwards and calculating the gradient of the error function with respect to the weights	Descending down the cost function until the minimum point and find the corresponding weights

4.4. TUNING HIDDEN-LAYER COUNT AND NEURON COUNT

As with learning rate and batch size, the number of hidden layers you add to your neural network is also a hyperparameter.

When it comes to the hidden layers, the main concerns are how many hidden layers and how many neurons are required?

Table 5.1: Determining the Number of Hidden Layers

Number of Hidden Layers	Result
none	Only capable of representing linear separable functions or decisions.
1	Can approximate any function that contains a continuous mapping from one finite space to another.
2	Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.

There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers, such as the following:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be $2/3$ the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

Moreover, the number of neurons and number layers required for the hidden layer also depends upon training cases, amount of outliers, the complexity of, data that is to be learned, and the type of activation functions used.

Most of the problems can be solved by using a single hidden layer with the number of neurons equal to the mean of the input and output layer. If less number of neurons is chosen it will lead to underfitting and high statistical bias. Whereas if we choose too many neurons it may lead to overfitting, high variance, and increases the time it takes to train the network.

4.5. AN INTERMEDIATE NET IN KERAS

The first few stages of our *Intermediate Net in Keras* Jupyter notebook are identical to those of its *Shallow Net* predecessor. We load the same Keras dependencies, load the MNIST dataset in the same way, and preprocess the data in the same way. As shown in Example 8.1, the situation begins to get interesting when we design our neural network architecture.

Example 8.1 Keras code to architect an intermediate-depth neural network

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

In addition to changes to the model architecture, we've also made changes to the parameters we specify when compiling our model, as shown in Example 8.2.

Example 8.2 Keras code to compile our intermediate-depth neural network

```
model.compile(loss='categorical_crossentropy',  
              optimizer=SGD(lr=0.1),  
              metrics=['accuracy'])
```

With these lines from Example 8.2, we:

- Set our loss function to cross-entropy cost by using `loss='categorical_crossentropy'` (in *Shallow Net in Keras*, we used quadratic cost by using `loss='mean_squared_error'`)
- Set our cost-minimizing method to stochastic gradient descent by using `optimizer=SGD`
- Specify our SGD learning rate hyperparameter η by setting `lr=0.1`
- Indicate that, in addition to the Keras default of providing feedback on loss, by setting `metrics=['accuracy']`, we'd also like to receive feedback on model accuracy

Finally, we train our intermediate net by running the code in Example 8.3.

Example 8.3 Keras code to train our intermediate-depth neural network

```
model.fit(X_train, y_train,  
          batch_size=128, epochs=20,  
          verbose=1,  
          validation_data=(X_valid, y_valid))
```

Chapter 5: Improving Deep Networks

5.1. Weight Initialization

- Xavier Glorot Distributions

Ans:

Weight initialization is an important design choice when developing deep learning neural network models.

Historically, weight initialization involved using small random numbers, although over the last decade, more specific heuristics have been developed that use information, such as the type of activation function that is being used and the number of inputs to the node.

These more tailored heuristics can result in more effective training of neural network models using the stochastic gradient descent optimization algorithm.

Each time, a neural network is initialized with a different set of weights, resulting in a different starting point for the optimization process, and potentially resulting in a different final set of weights with different performance characteristics.

Historically, weight initialization follows simple heuristics, such as:

- Small random values in the range $[-0.3, 0.3]$
- Small random values in the range $[0, 1]$
- Small random values in the range $[-1, 1]$

These heuristics continue to work well in general.

Weight Initialization for Sigmoid and Tanh

The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called “*glorot*” or “*xavier*” initialization

There are two versions of this weight initialization method, which we will refer to as “*xavier*” and “*normalized xavier*.”

Both approaches were derived assuming that the activation function is linear, nevertheless, they have become the standard for nonlinear activation functions like Sigmoid and Tanh, but not ReLU.

Xavier Weight Initialization

The xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range $-(1/\sqrt{n})$ and $1/\sqrt{n}$, where n is the number of inputs to the node.

- $\text{weight} = U[-(1/\sqrt{n}), 1/\sqrt{n}]$

We can implement this directly in Python.

The example below assumes 10 inputs to a node, then calculates the lower and upper bounds of the range and calculates 1,000 initial weight values that could be used for the nodes in a layer or a network that uses the sigmoid or tanh activation function.

After calculating the weights, the lower and upper bounds are printed as are the min, max, mean, and standard deviation of the generated weights.

The complete example is listed below.

```

# example of the xavier weight initialization
from math import sqrt
from numpy import mean
from numpy.random import rand
# number of nodes in the previous layer
n = 10
# calculate the range for the weights
lower, upper = -(1.0 / sqrt(n)), (1.0 / sqrt(n))
# generate random numbers
numbers = rand(1000)
# scale to the desired range
scaled = lower + numbers * (upper - lower)
# summarize
print(lower, upper)
print(scaled.min(), scaled.max())
print(scaled.mean(), scaled.std())

```

Running the example generates the weights and prints the summary statistics.

We can see that the bounds of the weight values are about -0.316 and 0.316. These bounds would become wider with fewer inputs and more narrow with more inputs.

We can see that the generated weights respect these bounds and that the mean weight value is close to zero with the standard deviation close to 0.17.

```

-0.31622776601683794 0.31622776601683794
-0.3157663248679193 0.3160839282916222
0.006806069733149146 0.17777128902976705

```

It can also help to see how the spread of the weights changes with the number of inputs.

For this, we can calculate the bounds on the weight initialization with different numbers of inputs from 1 to 100 and plot the result.

The complete example is listed below.

```

# plot of the bounds on xavier weight initialization for different numbers of inputs
from math import sqrt
from matplotlib import pyplot
# define the number of inputs from 1 to 100
values = [i for i in range(1, 101)]

```



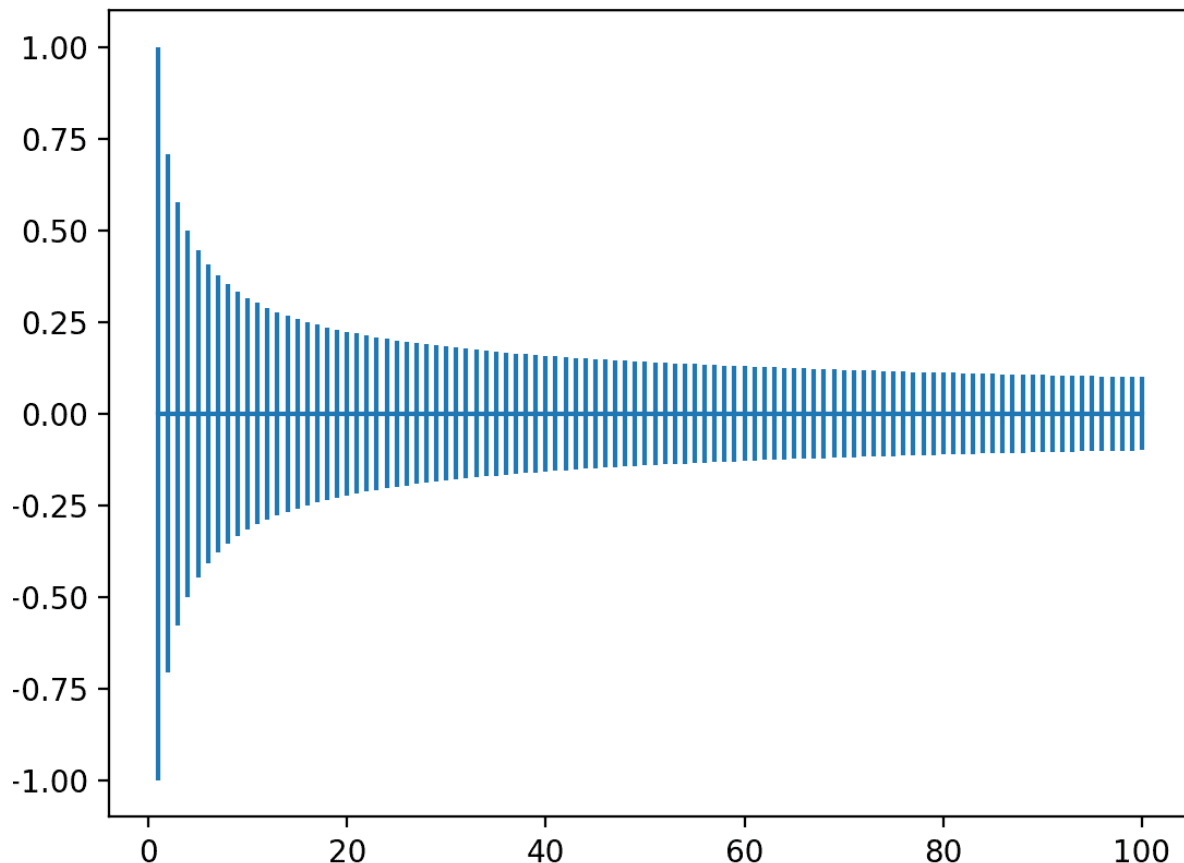
```
# calculate the range for each number of inputs
results = [1.0 / sqrt(n) for n in values]

# create an error bar plot centered on 0 for each number of inputs
pyplot.errorbar(values, [0.0 for _ in values], yerr=results)

pyplot.show()
```

Running the example creates a plot that allows us to compare the range of weights with different numbers of input values.

We can see that with very few inputs, the range is large, such as between -1 and 1 or -0.7 to -7. We can then see that our range rapidly drops to about 20 weights to near -0.1 and 0.1, where it remains reasonably constant.



5.2. Unstable Gradients

- Vanishing Gradients
- Exploding Gradients
- Batch Normalization

Ans:

Another issue associated with artificial neural networks, and one that becomes especially perturbing as we add more hidden layers, is *unstable gradients*. Unstable gradients can either be vanishing or explosive in nature.

Unstable gradients are a problem that can occur during the training of artificial neural networks. They can be either **vanishing** or **exploding** in nature.

Vanishing –

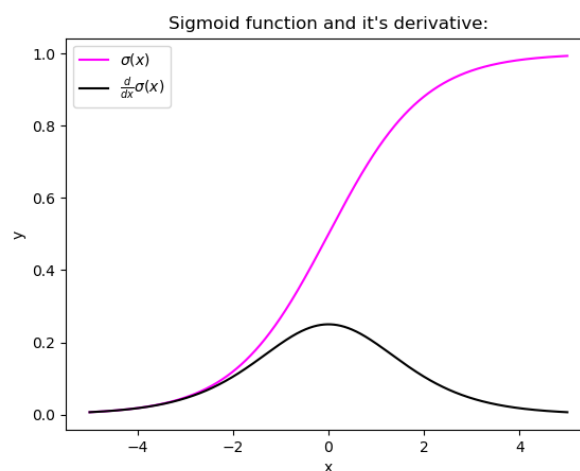
As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the **vanishing gradients** problem.

Exploding –

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the **exploding gradients** problem.

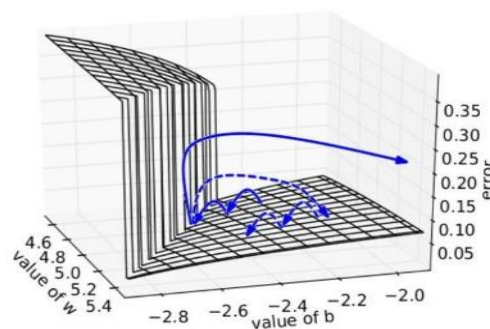
Why do the gradients even vanish/explode?

Certain activation functions, like the logistic function (sigmoid), have a very huge difference between the variance of their inputs and the outputs. In simpler words, they shrink and transform a larger input space into a smaller output space that lies between the range of [0,1].



Observing the above graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.

Similarly, in some cases suppose the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network. The parameters can sometimes become so large that they overflow and result in NaN values.



How to know if our model is suffering from the Exploding/Vanishing gradient problem?

Following are some signs that can indicate that our gradients are exploding/vanishing :

Exploding

There is an exponential growth in the model parameters.

The model weights may become NaN during training.

The model experiences avalanche learning.

Vanishing

The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).

The model weights may become 0 during training.

The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

Certainly, neither do we want our signal to explode or saturate nor do we want it to die out. The signal needs to flow properly both in the forward direction when making predictions as well as in the backward direction while calculating gradients.

Solutions

Now that we are well aware of the vanishing/exploding gradients problems, it's time to learn some techniques that can be used to fix the respective problems.

1. Proper Weight Initialization
2. Using Non-saturating Activation Functions
3. Batch Normalization
4. Gradient Clipping

Batch normalization is a technique that can be used to mitigate the problem of unstable gradients. Batch normalization works by normalizing the activations of each layer in the network, which helps to keep the gradients within a reasonable range.

Here is a diagram that illustrates how batch normalization works:

Input layer -> Batch normalization layer -> Hidden layer 1 -> Batch normalization layer -> Hidden layer 2 -> Batch normalization layer -> Output layer

The batch normalization layer in each hidden layer normalizes the activations of that layer to have a mean of 0 and a variance of 1. This helps to keep the gradients within a reasonable range, which makes it easier for the network to learn.

Batch normalization has been shown to be very effective at mitigating the problem of unstable gradients, and is now widely used in the training of deep neural networks.

Benefits of batch normalization

- Reduces the problem of unstable gradients
- Improves the convergence speed of the network
- Reduces the need for regularization techniques
- Improves the generalization performance of the network

The Following key points explain the intuition behind BN and how it works:

- It consists of adding an operation in the model just before or after the activation function of each hidden layer.
- This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
- In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs.
- To zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation.
- It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "Batch Normalization").

```
model = keras.models.Sequential([keras.layers.Flatten(input_shape=[28, 28]),
keras.layers.BatchNormalization(),
keras.layers.Dense(300, activation="relu"),
keras.layers.BatchNormalization(),
keras.layers.Dense(100, activation="relu"),
keras.layers.BatchNormalization(),
keras.layers.Dense(10, activation="softmax")])
```

we just added batch normalization after each layer (dataset : FMNIST)

```
model.summary()
```

📄 Model: "sequential_3"

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
batch_normalization (Batch Normalization)	(None, 784)	3136
dense_9 (Dense)	(None, 300)	235500
batch_normalization_1 (Batch Normalization)	(None, 300)	1200
dense_10 (Dense)	(None, 100)	30100
batch_normalization_2 (Batch Normalization)	(None, 100)	400
dense_11 (Dense)	(None, 10)	1010
Total params: 271,346		
Trainable params: 268,978		
Non-trainable params: 2,368		

5.3. Model Generalization — Avoiding Overfitting

- L1 and L2 Regularization
- Dropout
- Data Augmentation

Ans:

Overfitting relates to instances where the model tries to match non-existent data. This occurs when dealing with highly complex models where the model will match almost all the given data points and perform well in training datasets. However, the model would not be able to generalize the data point in the test data set to predict the outcome accurately.

What is Regularization?

It is a technique to prevent the model from overfitting by adding extra information to it.

This technique can be used in such a way that it will allow to maintain all variables or features in the model by reducing the magnitude of the variables. Hence, it maintains accuracy as well as a generalization of the model.

It mainly regularizes or reduces the coefficient of features toward zero. In simple words, "*In regularization technique, we reduce the magnitude of the features by keeping the same number of features.*"

There are two main types of regularization techniques: Ridge Regularization and Lasso Regularization.

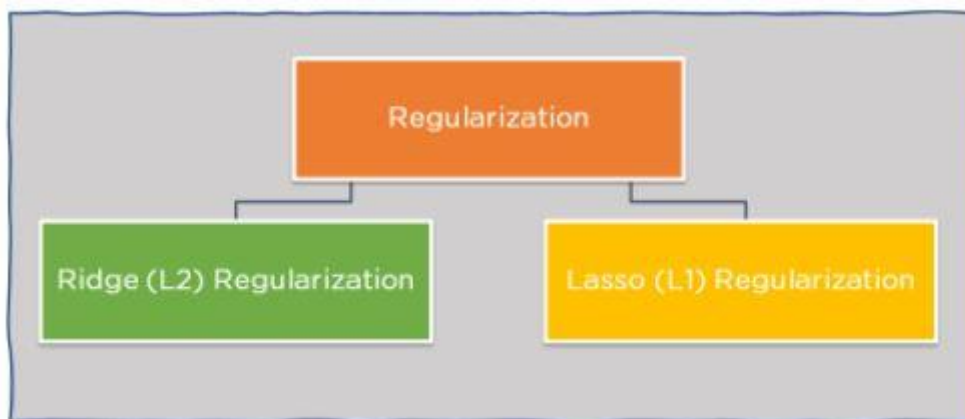


Figure 6: Regularization techniques

There are many different regularization techniques, but some of the most common ones include:

- **L1 regularization** (also known as Lasso regularization) adds a penalty to the sum of the absolute values of the model weights. This encourages the model to have fewer weights, which can help to prevent overfitting.
- **L2 regularization** (also known as Ridge regularization) adds a penalty to the sum of the squared values of the model weights. This also encourages the model to have fewer weights, but it is less aggressive than L1 regularization.
- **Elastic net regularization** is a combination of L1 and L2 regularization. It can be used to achieve a balance between the two approaches.
- **Dropout** is a technique that randomly drops out (sets to zero) some of the units in a neural network during training. This helps to prevent the network from becoming too reliant on any particular set of units.

Here are some examples of how regularization can be used in machine learning:

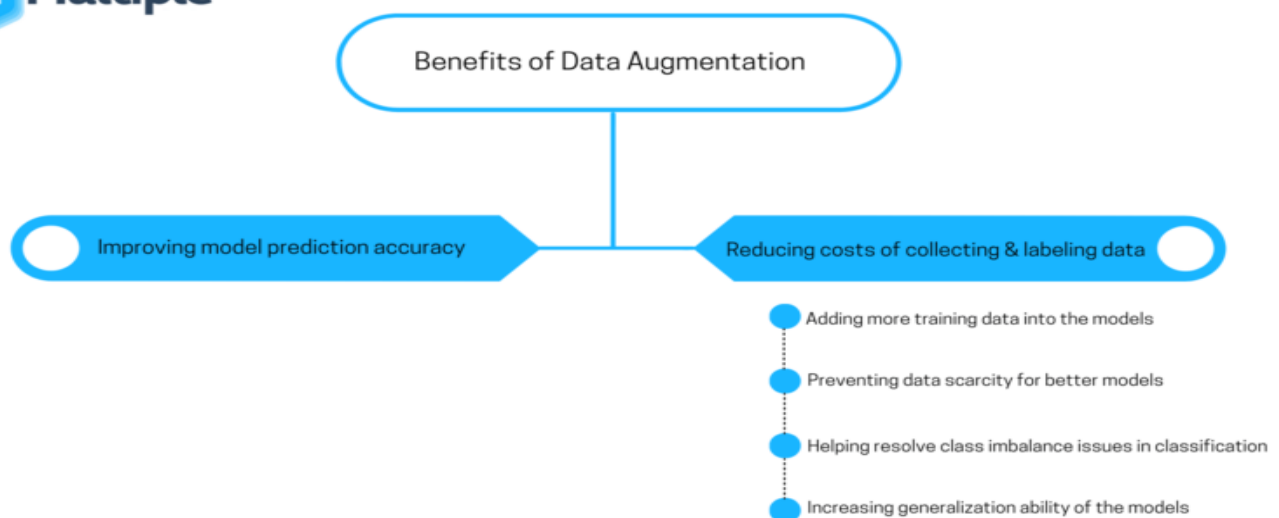
- In linear regression, regularization can be used to prevent the model from overfitting the training data. This can be done by adding a penalty to the sum of the squared values of the model weights.
- In logistic regression, regularization can be used to prevent the model from becoming too sensitive to noise in the training data. This can be done by adding a penalty to the sum of the absolute values of the model weights.
- In neural networks, regularization can be used to prevent the network from becoming too complex and overfitting the training data. This can be done by using L1 or L2 regularization, or by using dropout.

Regularization Technique	Penalty Function	Sparse Weight Vector
L1 regularization	Sum of the absolute values of the weights	Yes
L2 regularization	Sum of the squared values of the weights	No
Elastic net regularization	Combination of L1 and L2 regularization	Can be sparse or non-sparse

What is data augmentation?

Data augmentation is a set of techniques to artificially increase the amount of data by generating new data points from existing data. This includes making small changes to data or using deep learning models to generate new data points.

Data augmentation can be used to improve the performance of machine learning models by making them more robust to noise and variations in the data.



- ☞ Typically, More data = better learning
- ☞ Works well for image classification / object recognition tasks
- ☞ Also shown to work well for speech
- ☞ For some tasks it may not be clear how to generate such data

Dropout

- Dropout regularization is a technique for preventing neural networks from overfitting.
- Dropout works by randomly dropping out (setting to zero) a certain percentage of the neurons during training.
- Dropout forces the network to learn to rely on the remaining neurons, which makes it less likely to overfit the training data.
- Dropout can be used with any type of neural network.
- Dropout is a simple and effective way to prevent overfitting.

5.4. Fancy Optimizers **OR** different Optimizers for neural networks

- Momentum
- Nesterov Momentum
- AdaGrad
- AdaDelta and RMSProp
- Adam

Ans:

Gradient Descent

Gradient descent is an iterative machine learning optimization algorithm to reduce the cost function. This will help models to make accurate predictions.

Gradient indicates the direction of increase. As we want to find the minimum point in the valley we need to go in the opposite direction of the gradient. **We update parameters in the negative gradient direction to minimize the loss.**

$$\theta = \theta - \eta \nabla J(\theta; x, y)$$

θ is the weight parameter, η is the learning rate and $\nabla J(\theta; x, y)$ is the gradient of weight parameter θ

Types of Gradient Descent

Different types of Gradient descents are

- **Batch Gradient Descent or Vanilla Gradient Descent**
- **Stochastic Gradient Descent**
- **Mini batch Gradient Descent**

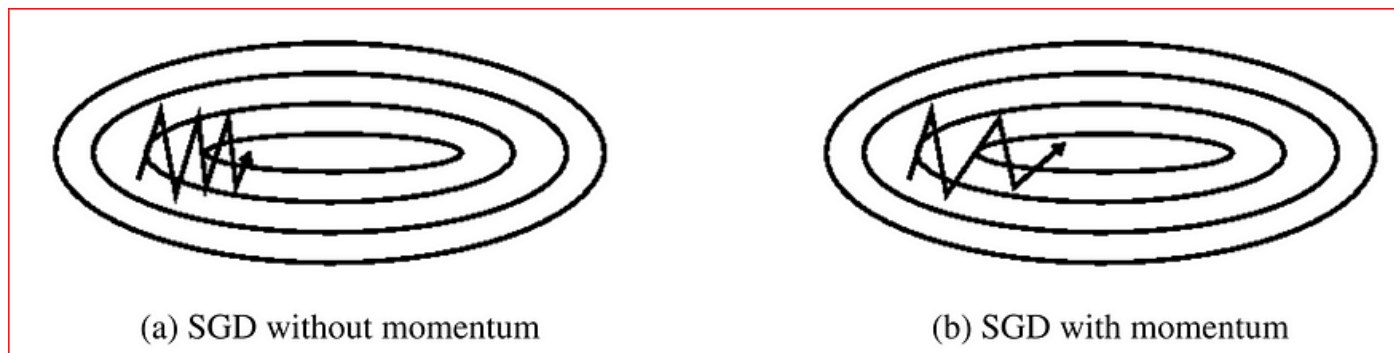
Role of an optimizer

Optimizers update the weight parameters to minimize the loss function. Loss function acts as guides to the terrain telling optimizer if it is moving in the right direction to reach the bottom of the valley, the global minimum.

Types of Optimizers

1. Momentum

Momentum is like a ball rolling downhill. The ball will gain momentum as it rolls down the hill.



Momentum helps accelerate Gradient Descent (GD) when we have surfaces that curve more steeply in one direction than in another direction. It also dampens the oscillation as shown above

For updating the weights it takes the gradient of the current step as well as the gradient of the previous time steps. This helps us move faster towards convergence.

Convergence happens faster when we apply momentum optimizer to surfaces with curves.

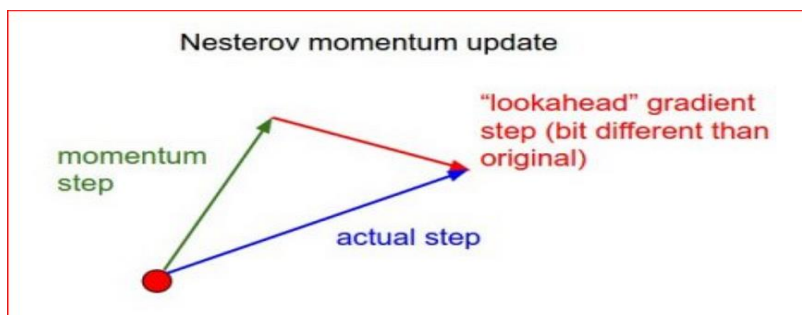
$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta; x, y)$$
$$\theta = \theta - v_t$$

Momentum Gradient descent takes gradient of previous time steps into consideration

2. Nesterov accelerated gradient (NAG)

Nesterov acceleration optimization is like a ball rolling down the hill but knows exactly when to slow down before the gradient of the hill increases again.

We calculate the gradient not with respect to the current step but with respect to the future step. We evaluate the gradient of the looked ahead and based on the importance then update the weights.



NAG is like you are going down the hill where we can look ahead in the future. This way we can optimize our descent faster. Works slightly better than standard Momentum.

$$\theta = \theta - v_t$$

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta - \gamma v_{t-1})$$

$\theta - \gamma v_{t-1}$ is the gradient of looked ahead

3. Adagrad — Adaptive Gradient Algorithm

We need to tune the learning rate in Momentum and NAG which is an expensive process.

Adagrad is an adaptive learning rate method. In Adagrad we adopt the learning rate to the parameters. We perform larger updates for infrequent parameters and smaller updates for frequent parameters.

It is well suited when we have sparse data as in large scale neural networks. GloVe word embedding uses adagrad where infrequent words required a greater update and frequent words require smaller updates.

For SGD, Momentum, and NAG we update for all parameters ϑ at once. We also use the same learning rate η . In Adagrad we use different learning rate for every parameter ϑ for every time step t

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \cdot g_t$$

G_t is sum of the squares of the past gradients w.r.t. to all parameters θ

Adagrad eliminates the need to manually tune the learning rate.

In the denominator, we accumulate the sum of the square of the past gradients. Each term is a positive term so it keeps on growing to make the learning rate η infinitesimally small to the point that algorithm is no longer able learning. **Adadelata, RMSProp, and adam tries to resolve Adagrad's radically diminishing learning rates.**

4. Adadelata

- Adadelata is an extension of Adagrad and it also tries to reduce Adagrad's aggressive, monotonically reducing the learning rate
- It does this by **restricting the window of the past accumulated gradient to some fixed size of w . Running average at time t then depends on the previous average and the current gradient**
- In Adadelata we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g_t]} \cdot g_t$$

5.RMSProp

- RMSProp is Root Mean Square Propagation. It was devised by Geoffrey Hinton.
- RMSProp tries to resolve Adagrad's radically diminishing learning rates by **using a moving average of the squared gradient**. It utilizes the magnitude of the recent gradient descents to normalize the gradient.
- In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.
- RMSProp divides the learning rate by the average of the exponential decay of squared gradients

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{(1 - \gamma)g^2_{t-1} + \gamma g_t + \epsilon}} \cdot g_t$$

γ is the decay term that takes value from 0 to 1. g_t is moving average of squared gradients

6.Adam — Adaptive Moment Estimation

- Another method that **calculates the individual adaptive learning rate for each parameter from estimates of first and second moments of the gradients**.
- It also reduces the radically diminishing learning rates of Adagrad
- Adam can be viewed as a **combination of Adagrad, which works well on sparse gradients and RMSprop which works well in online and nonstationary settings**.
- **Adam implements the exponential moving average of the gradients to scale the learning rate instead of a simple average as in Adagrad. It keeps an exponentially decaying average of past gradients**
- Adam is computationally efficient and has very little memory requirement
- **Adam optimizer is one of the most popular gradient descent optimization algorithms**

Adam algorithm first updates the exponential moving averages of the gradient(m_t) and the squared gradient(v_t) which is the estimates of the first and second moment.

Hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages as shown below

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of first and second moment respectively

Moving averages are initialized as 0 leading to moment estimates that are biased around 0 especially during the initial timesteps. This initialization bias can be easily counteracted resulting in bias-corrected estimates

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

\hat{m}_t and \hat{v}_t are bias corrected estimates of first and second moment respectively

Finally, we update the parameter as shown below

$$\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

Nadam- Nesterov-accelerated Adaptive Moment Estimation

- Nadam combines NAG and Adam
- Nadam is employed for noisy gradients or for gradients with high curvatures
- The learning process is accelerated by summing up the exponential decay of the moving averages for the previous and current gradient

5.5. A Deep Neural Network in Keras

The following is the procedure to implement Deep Neural Network using Keras

1. We load and preprocess the MNIST data

```
from keras.layers import Dropout
```

```
from keras.layers.normalization import BatchNormalization
```

2. Build the network

```
model = Sequential()
```

```
model.add(Dense(64, activation='relu', input_shape=(784,)))
```

```
model.add(BatchNormalization())
```

```
model.add(Dense(64, activation='relu'))
```

```
model.add(BatchNormalization())
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

3. Compile the model

```
model.compile(loss='categorical_crossentropy',
optimizer='adam',
metrics=['accuracy'])
```

5.6 Regression

a) importing required Regression model dependencies

```
from keras.datasets import boston_housing
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers.normalization import BatchNormalization
```

b) Loading the data is as simple as with the MNIST digits:

```
(X_train, y_train), (X_valid, y_valid) = boston_housing.load_data()
```

c) Regression model network architecture

```
model = Sequential()
model.add(Dense(32, input_dim=13, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(16, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Dense(1, activation='linear'))
```

4. Compiling a regression model

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

5. Fitting a regression model

```
model.fit(X_train, y_train, batch_size=8, epochs=32, verbose=1, validation_data=(X_valid, y_valid))
```

6. Predicting the median house price in a particular suburb of Boston

```
model.predict(np.reshape(X_valid[42], [1, 13]))
```


5.7. TENSORBOARD

When evaluating the performance of your model epoch over epoch, it can be tedious and time-consuming to read individual results numerically, as we did after running the code in, particularly if the model has been training for many epochs. Instead, TensorBoard is a convenient, graphical tool for:

- Visually tracking model performance in real time
- Reviewing historical model performances
- Comparing the performance of various model architectures and hyperparameter settings applied to fitting the same data

