UNIT I

Fundamentals of Deep Learning: Artificial Intelligence, History of Machine learning: Probabilistic Modeling, Early Neural Networks, Kernel Methods, Decision Trees, Random forests and Gradient Boosting Machines

Fundamentals of Machine Learning: Four Branches of Machine Learning, Evaluating Machine learning Models, Overfitting and Underfitting

……

1. Artificial Intelligence

- The topic of whether computers might be taught to "think" was first posed in the 1950s by a small group of pioneers in the developing discipline of computer science. The implications of this question are still being researched today.
- The endeavour to automate intellectual processes typically carried out by humans would serve as a succinct explanation of the area. As a result, AI is a broad area that comprises a variety of methods that include learning as well as machine learning and deep learning. For instance, early chess programmes used just hardcoded rules created by programmers and were not machine learning applications.
- For many years, experts thought that AI could be achieved by having programmers write a lot of rules for computers to follow.
- This approach is called symbolic AI, and it was the main way of doing AI from the 1950s to the late 1980s.
- Symbolic AI reached its peak popularity in the 1980s, when expert systems were very popular.

1.1. Artificial intelligence, machine learning, and deep learning

Artificial intelligence (AI), machine learning (ML), and deep learning (DL) are all terms that are often used interchangeably, but they actually have different meanings.

- **Artificial intelligence** is a broad term that refers to the ability of machines to perform tasks that are typically associated with human intelligence, such as learning, reasoning, and problem-solving.
- **Machine learning** is a subset of AI that involves the development of algorithms that can learn from data without being explicitly programmed. Machine learning algorithms are trained on large datasets, and they can then be used to make predictions or decisions about new data.
- **Deep learning** is a subset of machine learning that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain, and they can be used to solve complex problems that would be difficult or impossible to solve with traditional machine learning algorithms.

In other words, AI is the umbrella term, ML is a subset of AI, and DL is a subset of ML.

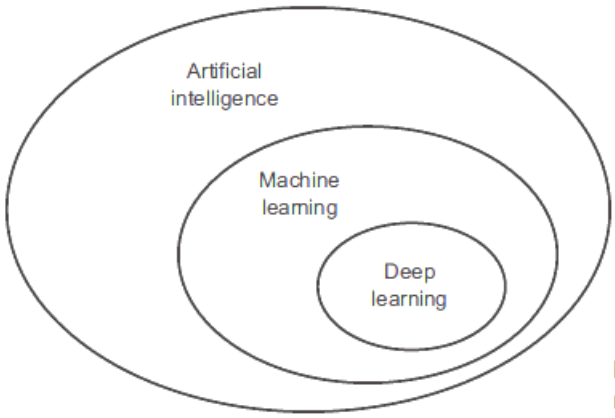


Figure 1.1 Artificial intelligence, machine learning, and deep learning

Here are some examples of how AI, ML, and DL are being used today:

- **AI** is being used to develop self-driving cars, facial recognition software, and spam filters.
- **ML** is being used to predict customer behaviour, optimize product recommendations, and personalize marketing campaigns.
- **DL** is being used to develop natural language processing (NLP) models, image recognition algorithms, and medical diagnosis tools.

AI, ML, and DL are all rapidly growing fields, and they are having a major impact on our lives. As these technologies continue to develop, we can expect to see even more innovative and groundbreaking applications in the years to come.

1.2 Machine Learning

Machine learning is a type of artificial intelligence (AI) that allows software applications to become more accurate in predicting outcomes without being explicitly programmed to do so. Machine learning algorithms use historical data as input to predict new output values.

In other words, machine learning algorithms can learn from data and improve their performance over time. This is in contrast to traditional programming, where software applications are explicitly programmed to perform specific tasks.

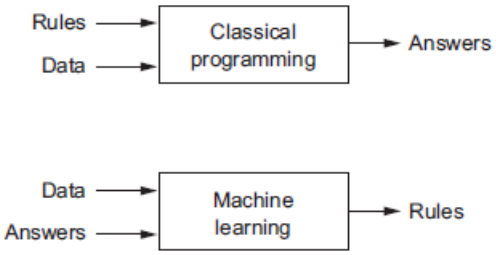


Figure 1.2 Machine learning: a new programming paradigm

Machine learning is used in a wide variety of applications, including:

- **Predictive analytics** - Machine learning can be used to predict future events, such as customer behavior, product demand, and fraud.
- **Personalization** - Machine learning can be used to personalize user experiences, such as product recommendations and advertising.
- **Fraud detection** - Machine learning can be used to detect fraudulent transactions, such as credit card fraud and insurance fraud.
- **Medical diagnosis** - Machine learning can be used to diagnose medical conditions, such as cancer and heart disease.

Here are some examples of how machine learning is used today:

- **Netflix** uses machine learning to recommend movies and TV shows to its users.
- **Amazon** uses machine learning to recommend products to its customers.
- **Google** uses machine learning to power its search engine, translate languages, and recognize objects in images.
- **Spotify** uses machine learning to create personalized playlists for its users.
- **Tesla** uses machine learning to power its self-driving cars.

1.3 The “deep” in deep learning

Deep learning is a type of machine learning that learns from data by creating successive layers of increasingly meaningful representations. The depth of a deep learning model refers to how many layers it has. Modern deep learning models often have tens or even hundreds of layers.

In contrast, other approaches to machine learning typically only learn one or two layers of representations. These approaches are sometimes called shallow learning.

The main difference between deep learning and shallow learning is that deep learning models can learn more complex representations of data. This makes them more powerful and able to solve more difficult problems.

Here is an analogy that might help you understand deep learning:

Imagine you are trying to understand a book. You could start by reading the first page, then the second page, and so on. This would be like shallow learning. You would only be able to understand the book at a superficial level.

However, you could also read the book by first reading the introduction, then the table of contents, then the chapters in order. This would be like deep learning. You would be able to understand the book at a much deeper level.

Deep learning is a powerful technique that is having a major impact on many different fields. It is used in applications such as image recognition, natural language processing, and machine translation.

Neural networks are a type of mathematical model that is inspired by the human brain. They are made up of layers of interconnected nodes, and they can learn to represent complex patterns in data.

In deep learning, neural networks are used to learn successive layers of increasingly meaningful representations. This is done by feeding the network a large amount of data, and then adjusting the weights of the connections between the nodes until the network is able to correctly classify or predict the data.

The term "neural network" is a reference to neurobiology, but neural networks are not models of the brain. The brain is a complex organ, and we do not yet fully understand how it works. Neural networks are much simpler models, and they are not designed to be a complete representation of the brain.

The idea that deep learning is "just like the brain" is a popular misconception. There are some similarities between neural networks and the brain, but there are also many important differences. For example, the brain uses a different learning mechanism than neural networks, and it is able to learn from much less data.

In the context of deep learning, the term "neural network" is simply a way of referring to a particular type of mathematical model. It is not meant to imply that the model is a complete representation of the brain.

Example:

The representations learned by a deep-learning algorithm are typically **abstract** and **meaningful**. They are abstract in the sense that they are not directly related to the original data. For example, the representations learned by a deep-learning algorithm for image recognition are not simply the pixels of the image. Instead, they are more abstract features, such as the edges, shapes, and textures of the image.

The representations learned by a deep-learning algorithm are also meaningful in the sense that they are related to the task that the algorithm is trying to perform. For example, the representations learned by a deep-learning algorithm for image recognition are related to the identity of the digit in the image.

Let's take a look at how a network several layers deep (see Figure 1.5) transform an image of a digit in order to recognize what digit it is.

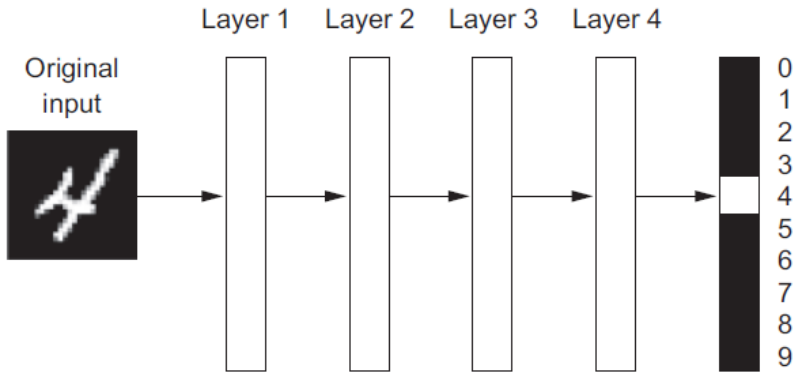


Figure 1.5 A deep neural network for digit classification

Figure 1.6 illustrates how the network alters the digit picture into representations that diverge ever more from the original and reveal even more about the outcome. A deep network can be compared to an information distillation process with multiple stages, where information is passed through progressively purer filters until it is suitable for a given purpose.

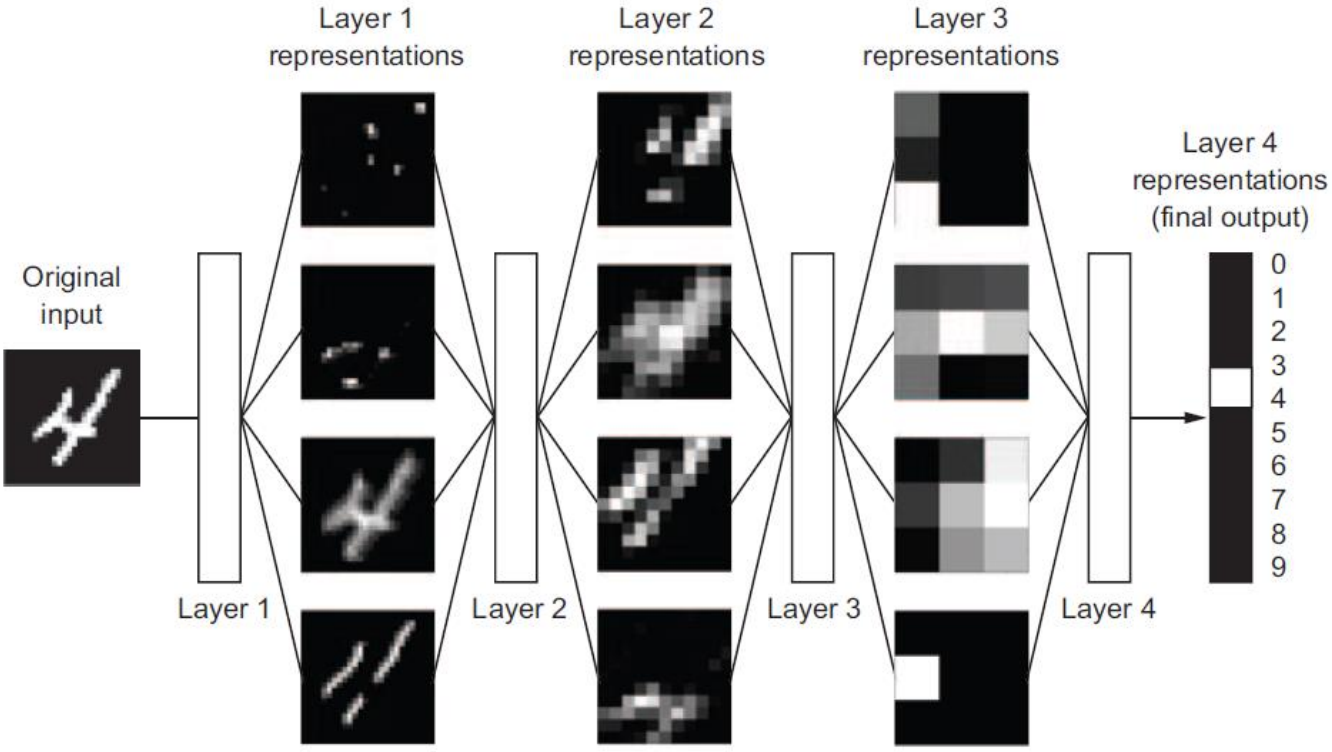


Figure 1.6 Deep representations learned by a digit-classification model

1.4 Understanding how deep learning works, in three figures

Machine learning is about finding a way to map inputs to targets. For example, we could map images of cats to the label "cat". We do this by observing many examples of inputs and targets, and then learning a sequence of simple data transformations that can be used to map new inputs to targets. These data transformations are learned by exposure to examples, which means that we show the machine learning algorithm many examples of inputs and targets, and it learns how to map the inputs to the targets.

Deep neural networks are a type of machine learning algorithm that can do this input-to-target mapping very well. They do this by using a deep sequence of simple data transformations, which allows them to learn very complex mappings.

Now let's look at how this learning happens, concretely.

- The weights of a layer in a deep neural network determine how the layer transforms its input data.
- The weights are essentially a bunch of numbers.
- Finding the correct values for all the weights is a daunting task.
- This is because modifying the value of one weight can affect the behavior of all the other weights in the network.
- The process of finding the correct values for the weights of a deep neural network is called training.
- Training involves feeding the network a large dataset of examples.
- The training process is typically done using an algorithm called backpropagation.

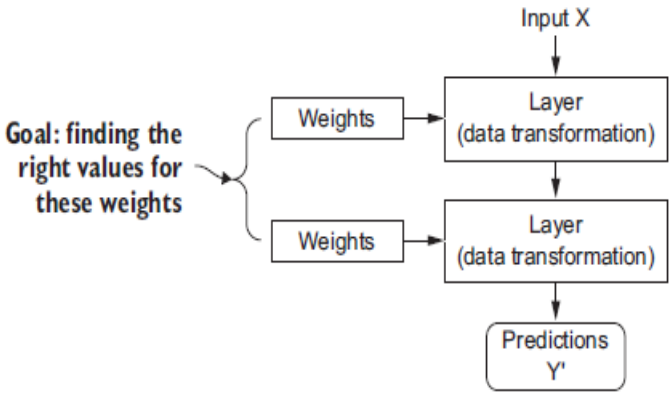


Figure 1.7 A neural network is parameterized by its weights.

To control something, first you need to be able to observe it. To control the output of a neural network, you need to be able to measure how far this output is from what you expected. This is the job of the *loss function* of the network, also called the *objective function*.

The loss function is a measure of how well the network has performed on a particular example. The loss function takes the predictions of the network and the true target, and then calculates a distance score. The distance score is a measure of how far the predictions are from the target.

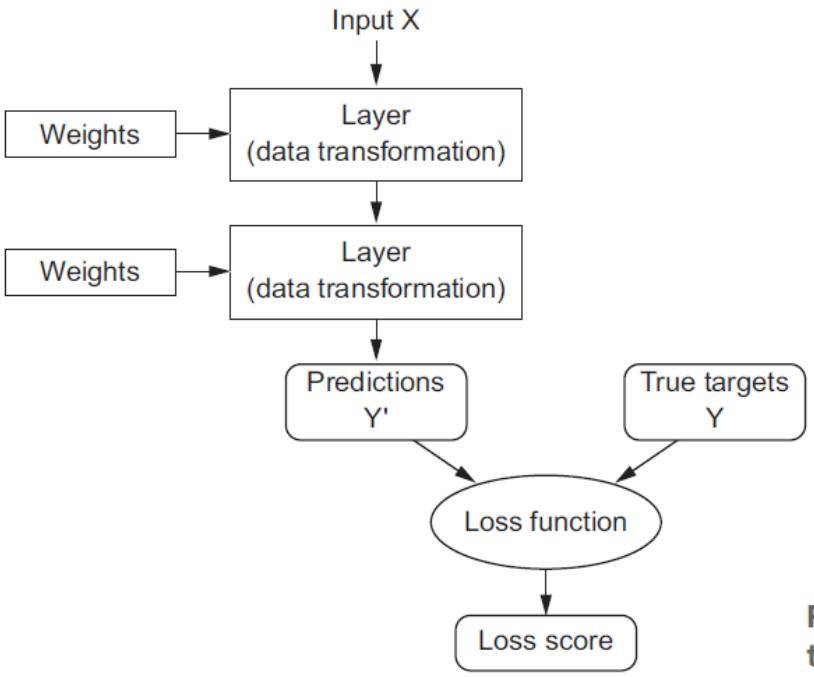


Figure 1.8 A loss function measures the quality of the network's output.

The loss function is used to train the neural network. The goal of training is to minimize the loss function. This means that the goal is to get the predictions of the network as close to the target as possible.

There are many different types of loss functions. Some common loss functions include:

- **Mean squared error (MSE)**: This is a loss function that measures the squared difference between the predictions and the target.
- **Cross-entropy**: This is a loss function that is used for classification problems. It measures the difference between the predicted probabilities and the true probabilities.
- **Huber loss**: This is a loss function that is less sensitive to outliers than MSE.

The choice of loss function depends on the type of problem that the neural network is being trained to solve.

The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example. This adjustment is the job of the *optimizer*, which implements what's called the *Backpropagation* algorithm: the central algorithm in deep learning.

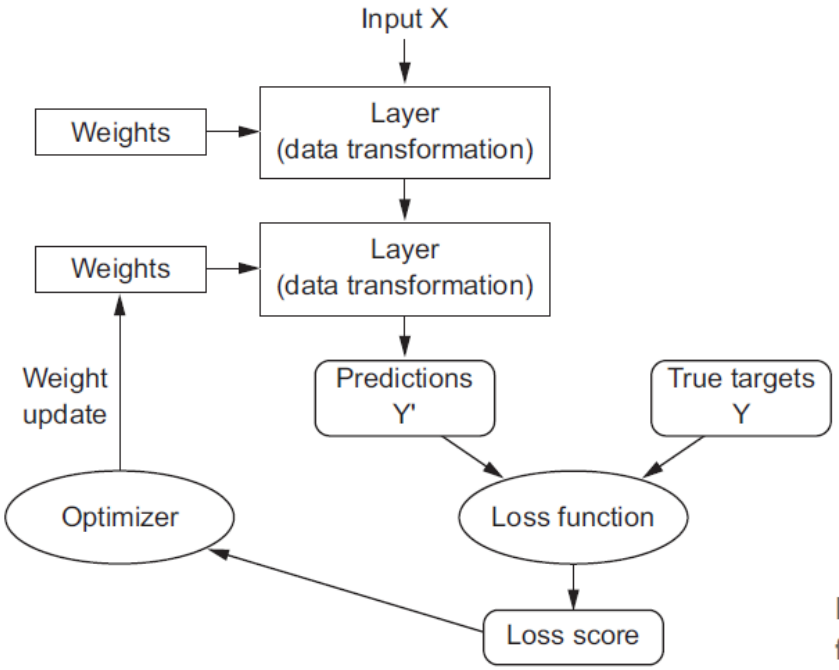


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

Initially, the weights of the network are assigned random values, so the network merely implements a series of random transformations.

Naturally, its output is far from what it should ideally be, and the loss score is accordingly very high. But with every example the network processes, the weights are adjusted a little in the correct direction, and the loss score decreases. This is the *training loop*, which, repeated a sufficient number of times (typically tens of iterations over thousands of examples), yields weight values that minimize the loss function. A network with a minimal loss is one for which the outputs are as close as they can be to the targets: a trained network.

1.5 *What deep learning has achieved so far*

In particular, deep learning has achieved the following breakthroughs, all in historically difficult areas of machine learning:

- Near-human-level image classification
- Near-human-level speech recognition
- Near-human-level handwriting transcription
- Improved machine translation
- Improved text-to-speech conversion
- Digital assistants such as Google Now and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, and Bing
- Improved search results on the web
- Ability to answer natural-language questions
- Superhuman Go playing

2. Before deep learning: a brief history of machine learning

Deep learning has become very popular and is getting a lot of attention and investment in the AI field. However, it's essential to know that it's not the only successful type of machine learning. Many other algorithms are used in the industry today, and they are not necessarily deep learning algorithms.

Deep learning might not always be the best choice for a particular task. Sometimes there might not be enough data for it to work well, or another algorithm could solve the problem more effectively. If you're new to machine learning and only know about deep learning, you might end up trying to use it for everything and see every problem as a "nail" for your "hammer."

To avoid this trap, it's crucial to learn about other approaches to machine learning and use them when they are more suitable for the task at hand.

2.1. Probabilistic modeling

Probabilistic modeling is a way to use statistics to understand data. It's been around for a long time, and it's still used a lot today. One of the most popular probabilistic models is called **Naive Bayes**.

Naive Bayes is a simple but powerful algorithm that can be used for a variety of tasks, including spam filtering, text classification, and medical diagnosis. It works by assuming that each feature in a dataset is independent of the others. This means that the probability of a certain outcome is only affected by the probability of that outcome for that feature.

Naive Bayes is a type of machine learning algorithm that uses Bayes' theorem to classify data. It assumes that the features in the data are independent of each other, which is why it's called Naive Bayes.

Bayes' theorem is a mathematical formula that can be used to calculate the probability of an event happening, given some prior knowledge. In the case of Naive Bayes, the prior knowledge is the distribution of features in the training data.

Naive Bayes is a simple algorithm, but it can be very effective for classification tasks. It's also fast and scalable, making it a good choice for large datasets.

A closely related model is the *logistic regression* (logreg for short), which is sometimes considered to be the "hello world" of modern machine learning.

2.2. Early neural networks

Early neural networks were inspired by the biological neural networks in the human brain. They were simple models that consisted of a few layers of interconnected neurons. These neurons were typically threshold gates, which means that they would only activate if their input was above a certain threshold.

The first early neural network was the perceptron, which was developed by Frank Rosenblatt in 1957. The perceptron was a simple model that could be used to classify binary data. It was a single-layer neural network with a threshold gate at the output.

Other early neural networks included the ADALINE and MADALINE networks, which were developed by Bernard Widrow and Marcian Hoff in 1959. These networks were also single-layer neural networks, but they used a different type of neuron called a linear unit.

Early neural networks were limited in their capabilities. They could only be used to solve simple problems, and they were not very robust to noise. However, they laid the foundation for the development of more powerful neural networks that are used today.

Here are some of the key limitations of early neural networks:

- They were only able to learn simple linear relationships.
- They were not very robust to noise.
- They were computationally expensive to train.

Despite these limitations, early neural networks were a significant breakthrough in the field of artificial intelligence. They showed that it was possible to build machine learning models that were inspired by the human brain. This led to the development of more powerful neural networks that are used today.

2.3. Kernel Methods

As neural networks started to gain some respect among researchers in the 1990s, thanks to this first success, a new approach to machine learning rose to fame and quickly sent neural nets back to oblivion: kernel methods

Kernel methods are a group of classification algorithms, the best known of which is the *support vector machine* (SVM).

SVMs aim at solving classification problems by finding good *decision boundaries* between two sets of points belonging to two different categories.

Figure 1.10
A decision boundary

A decision boundary can be thought of as a line or surface separating your training data into two spaces corresponding to two categories.

To classify new data points, you just need to check which side of the decision boundary they fall on.

SVMs proceed to find these boundaries in two steps:

1.The data is mapped to a new high-dimensional representation where the decision boundary can be expressed as a hyperplane (if the data was two dimensional, a hyperplane would be a straight line).

2. A good decision boundary (a separation hyperplane) is computed by trying to maximize the distance between the hyperplane and the closest data points from each class, a step called *maximizing the margin*. This allows the boundary to generalize well to new samples outside of the training dataset.

The technique of mapping data to a high-dimensional representation where a classification problem becomes simpler may look good on paper, but in practice it's often computationally intractable. That's where the *kernel trick* comes in (the key idea that kernel methods are named after).

The kernel trick is a technique used in machine learning that allows kernel methods to operate in a high-dimensional space without explicitly computing the coordinates of the data in that space. This can be computationally advantageous, as it can avoid the need to perform matrix multiplications in high-dimensional space.

2.4. Decision trees, random forests, and gradient boosting machines

Decision trees are flowchart-like structures that let you classify input data points or predict output values given inputs . They're easy to visualize and interpret.

Decisions trees learned from data began to receive significant research interest in the 2000s, and by 2010 they were often preferred to kernel methods.

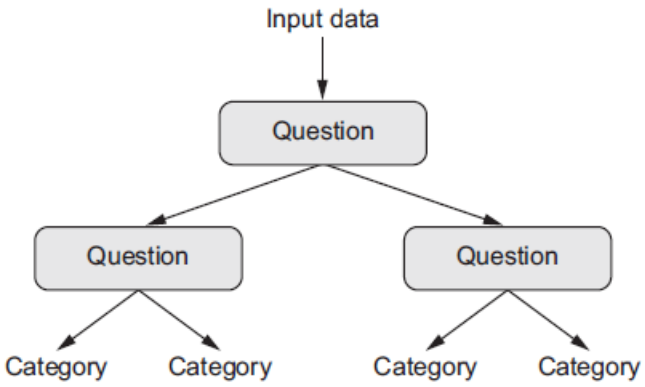


Figure 1.11 A decision tree: the parameters that are learned are the questions about the data. A question could be, for instance, "Is coefficient 2 in the data greater than 3.5?"

In particular, the *Random Forest* algorithm introduced a robust, practical take on decision-tree learning that involves building a large number of specialized decision trees and then ensembling their outputs. Random forests are applicable to a wide range of problems—you could say that they're almost always the second-best algorithm for any shallow machine-learning task.

A gradient boosting machine, much like a random forest, is a machine-learning technique based on ensembling weak prediction models, generally decision trees. It uses *gradient boosting*, a way to improve any machine-learning model by iteratively training new models that specialize in addressing the weak points of the previous models.

Applied to decision trees, the use of the gradient boosting technique results in models that strictly outperform random forests most of the time, while having similar properties. It may be one of the best, if not *the* best, algorithm for dealing with non-perceptual data today.

Chapter-II

Fundamentals of Machine Learning: Four Branches of Machine Learning, Evaluating Machine learning Models, Overfitting and Underfitting

.....

1. Four branches of machine learning

Machine learning is a subfield of artificial intelligence that focuses on developing algorithms and statistical models that enable computers to learn and improve their performance on a specific task through experience. There are four main branches of machine learning:

1. **Supervised Learning**: In supervised learning, the algorithm is trained on a labeled dataset, where each input data point is associated with the corresponding target or output label. The goal of the algorithm is to learn a mapping from inputs to outputs, enabling it to make predictions on new, unseen data. Common tasks in supervised learning include classification (assigning labels to input data) and regression (predicting numerical values).
2. **Unsupervised Learning**: Unsupervised learning involves training algorithms on an unlabeled dataset, where the data does not have predefined output labels. The goal of unsupervised learning is to discover patterns, structures, or relationships within the data. Clustering, where the algorithm groups similar data points together, and dimensionality reduction, which aims to simplify data while preserving essential characteristics, are examples of unsupervised learning tasks.
3. **Semi-Supervised Learning**: Semi-supervised learning is a combination of supervised and unsupervised learning. The algorithm is trained on a dataset that contains both labeled and unlabeled data. The labeled data provides some information for guidance, and the unlabeled data helps the algorithm learn more about the underlying structure of the data, often leading to better performance when labeled data is limited or expensive to obtain.
4. **Reinforcement Learning**: Reinforcement learning is different from the previous three types as it involves an agent that interacts with an environment to achieve a goal. The agent takes actions in the environment and receives feedback in the form of rewards or penalties. The goal of the agent is to learn a policy or strategy that maximizes the cumulative reward over time. Reinforcement learning is commonly used in applications such as game playing, robotics, and autonomous systems.

These four branches cover a broad range of machine learning techniques and applications, and they continue to evolve and advance as researchers and practitioners explore new algorithms and approaches.

Branch	Description	Examples
Supervised learning	The model is trained on labeled data.	Spam filtering, image classification, fraud detection
Unsupervised learning	The model is trained on unlabeled data.	Clustering, dimensionality reduction, anomaly detection
Semi-supervised learning	The model is trained on a combination of labeled and unlabeled data.	Natural language processing, medical diagnosis
Reinforcement learning	The model learns to behave in an environment by trial and error.	Game playing, robotics

2. Evaluating Machine learning Models

Evaluating machine learning models is an important step in the machine learning process. It allows you to assess the performance of your model and identify any areas where it can be improved. There are a number of different metrics that can be used to evaluate machine learning models, each with its own strengths and weaknesses.

2.1. Training, validation, and test sets

Evaluating a model always boils down to splitting the available data into three sets: training, validation, and test. You train on the training data and evaluate your model on the validation data. Once your model is ready for prime time, you test it one final time on the test data.

You may ask, why not have two sets: a training set and a test set? You would train on the training data and evaluate on the test data. Much simpler!

- **Overfitting.** If we only train on the training set, and then evaluate on the test set, there is a risk that the model will overfit the training data. This means that the model will learn the specific details of the training data, and will not be able to generalize to new data.
- **Bias.** If we only use the training set to evaluate the model, we may not be able to identify any biases in the model. This is because the training set is the only data that the model has seen, so it is possible that the model will learn the biases that are present in the training data.
- **Unreliability.** If we only use the test set to evaluate the model, then the results of the evaluation may not be reliable. This is because the test set is only a small sample of the data that the model will be used on in the real world.

To address these issues, we often split the data into three sets: a training set, a validation set, and a test set. The training set is used to train the model, the validation set is used to tune the hyperparameters of the model, and the test set is used to evaluate the final model.

This approach helps to prevent overfitting, identify biases, and ensure that the results of the evaluation are reliable.

In addition to the three sets mentioned above, some people also use a fourth set called the **holdout set**. The holdout set is a separate set of data that is not used for training, validation, or testing. **The holdout set is used to get an independent estimate of the model's performance.**

The use of multiple sets can be a bit more complex, but it is generally considered to be a best practice for building reliable machine learning models.

Splitting your data into training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it that can come in handy when little data is available.

Let us review three classic evaluation recipes: simple hold-out validation, K-fold validation, and iterated K-fold validation with shuffling

1. SIMPLE HOLD-OUT VALIDATION

Simple hold-out validation is a method of evaluating a machine learning model by splitting the data into two sets: a training set and a test set. The training set is used to train the model, and the test set is used to evaluate the model's performance.

The following steps are involved in simple hold-out validation:

1. Split the data into a training set and a test set. The training set should typically be about 80% of the data, and the test set should be about 20% of the data.
2. Train the model on the training set.

3. Evaluate the model on the test set.
4. Repeat steps 2 and 3 for different hyperparameters or different models.

The model with the best performance on the test set is the best model.

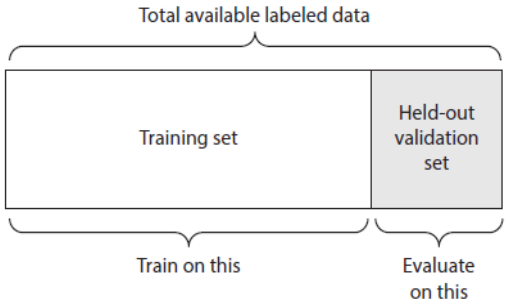


Figure 4.1 Simple hold-out validation split

Here are some of the advantages of simple hold-out validation:

- It is simple to understand and implement.
- It is a good way to evaluate the performance of a model.

Here are some of the disadvantages of simple hold-out validation:

- It can be sensitive to the way that the data is split into training and test sets.
- It can be difficult to get a reliable estimate of the model's performance if the test set is too small.

simple hold-out validation is a good way to evaluate the performance of a machine learning model. However, it is important to be aware of its limitations.

2. K-FOLD VALIDATION

K-fold validation is a method of evaluating a machine learning model by splitting the data into k folds. The model is trained k times, each time using a different fold as the validation set and the remaining folds as the training set. The performance metrics obtained from each fold are averaged to provide a more robust estimate of the model's generalization performance.

The following steps are involved in k-fold validation:

1. Split the data into k folds.
2. For each fold:
    - Train the model on the remaining k-1 folds.
    - Evaluate the model on the current fold.
3. Average the performance metrics from each fold.

The model with the best average performance is the best model.

Here are some of the advantages of k-fold validation:

- It is more robust to overfitting than simple hold-out validation.
- It can provide a more reliable estimate of the model's performance.

Here are some of the disadvantages of k-fold validation:

- It can be more computationally expensive than simple hold-out validation.
- It can be difficult to implement if the data is not evenly distributed.

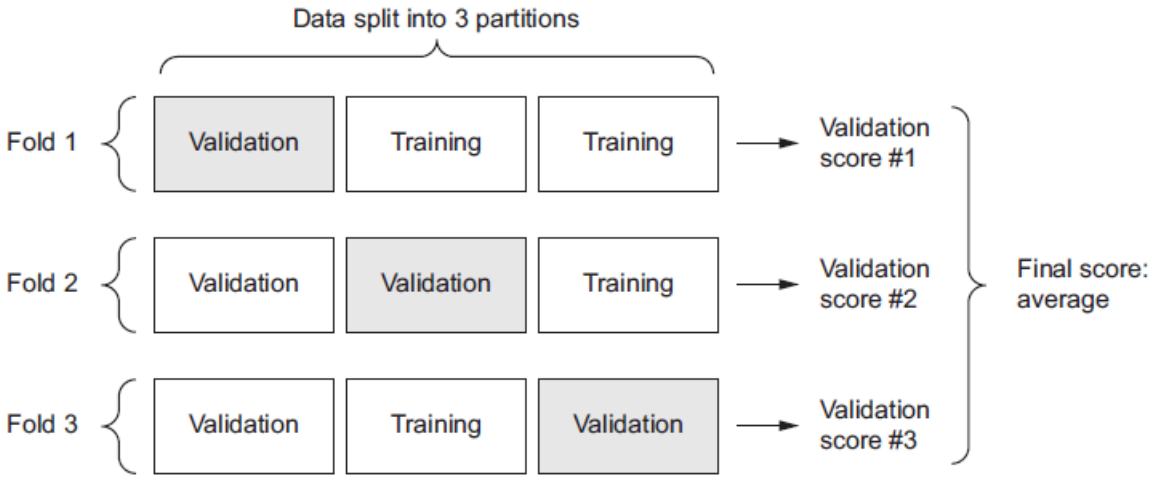


Figure 4.2 Three-fold validation

k-fold validation is a more sophisticated method of evaluating a machine learning model than simple hold-out validation. It can provide a more reliable estimate of the model's performance, but it can also be more computationally expensive.

2.2. *Things to keep in mind*

Keep an eye out for the following when you are choosing an evaluation protocol

- **The purpose of the evaluation.** What do you hope to achieve by evaluating your program or intervention? Are you looking to measure its effectiveness, efficiency, or impact? Once you know the purpose of the evaluation, you can choose a protocol that is designed to answer your specific questions.
- **The scope of the evaluation.** What aspects of your program or intervention will you be evaluating? Will you be looking at the overall impact, or will you be focusing on specific outcomes? The scope of the evaluation will determine the type of data you need to collect and the methods you will use to collect it.
- **The resources available.** How much time, money, and personnel do you have to dedicate to the evaluation? The resources available will affect the type of protocol you can choose. For example, if you have limited resources, you may need to choose a protocol that uses less data or that is less time-consuming to implement.
- **The ethical considerations.** What ethical issues are relevant to your evaluation? For example, if you are collecting personal data about participants, you will need to make sure that you have their consent and that you are protecting their privacy.

Once you have considered these factors, you can start to look for evaluation protocols that are a good fit for your needs. There are many different evaluation protocols available, so you should be able to find one that meets your specific requirements.

3. Overfitting and underfitting

Overfitting and underfitting are two common problems that can occur in machine learning. Overfitting occurs when a model learns the training data too well and starts to memorize the noise and outliers in the data. This can lead to the model performing poorly on new data that it has not seen before. Underfitting occurs when a model does not learn the training data well enough and is unable to make accurate predictions.

Here are some examples of overfitting and underfitting:

- **Overfitting:** A model that is trained to predict whether a patient has cancer might learn to memorize the specific features of the training data that are associated with cancer. This would allow the model to make accurate predictions on the training data, but it would also cause the model to perform poorly on new data that does not have the same features.
- **Underfitting:** A model that is trained to predict the price of a house might not learn the relationship between the features of the house and its price. This would cause the model to make inaccurate predictions on both the training data and new data

Optimization and generalization are two important concepts in machine learning. Optimization refers to the process of finding the best parameters for a model, while generalization refers to the ability of a model to make accurate predictions on new data.

The goal of machine learning is to build models that can generalize well to new data. However, there is a tension between optimization and generalization. If we optimize a model too much, it may become too complex and start to overfit the training data. This means that the model will make accurate predictions on the training data, but it will not generalize well to new data.

On the other hand, if we do not optimize the model enough, it may not learn the training data well enough and will not be able to make accurate predictions on new data.

The challenge in machine learning is to find a balance between optimization and generalization. This can be done by using regularization techniques, which add a penalty to the model's complexity. **Regularization** can help to prevent the model from overfitting the training data and improve its ability to generalize to new data.

Let us review some of the most common regularization techniques

3.1. Reducing the network's size

Reducing the size of a network can help to prevent overfitting by making the model less complex. A less complex model is less likely to memorize the noise and outliers in the training data. As a result, the model will be more likely to generalize well to new data.

3.2. Adding weight regularization

- Weight regularization is a technique used to prevent neural networks from overfitting the training data.
- There are two main types of weight regularization: L1 and L2.
- To add weight regularization to a neural network, you can use the kernel_regularizer argument when creating the layer.
- The amount of weight regularization to use is a hyperparameter that you will need to tune.

L1 regularization adds a penalty to the loss function that is proportional to the absolute value of the weights. This can help to reduce the number of non-zero weights in the model, which can be useful for feature selection.

L2 regularization adds a penalty to the loss function that is proportional to the square of the weights. This is the most common type of weight regularization and it is generally effective at preventing overfitting.

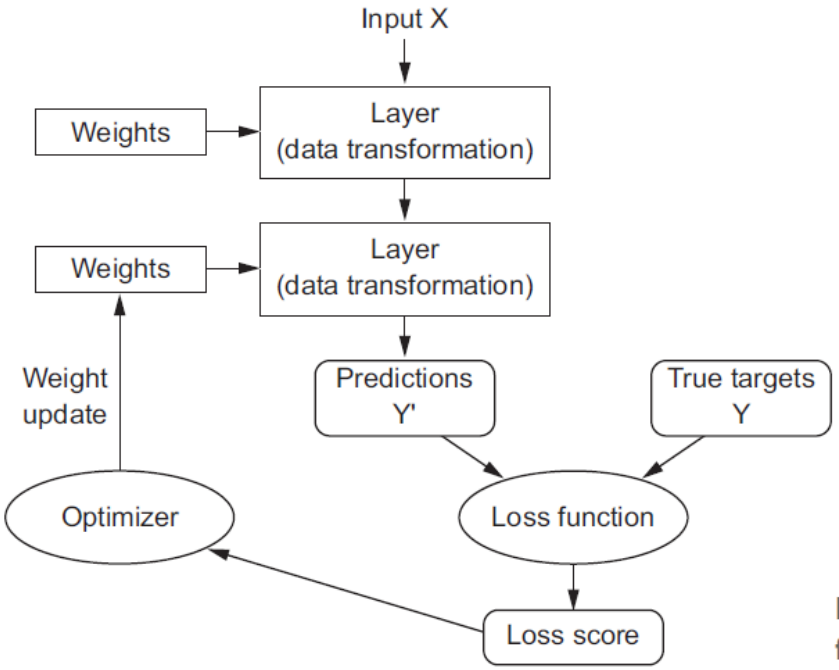


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

3.3.Adding dropout

Dropout is a regularization technique that can also be used to prevent neural networks from overfitting.

To add dropout to a neural network, you can use the Dropout layer.

For example, the following code adds dropout with a rate of 0.2 to the hidden layer of a neural network:

```
model.add(Dropout(0.2))
```

The rate argument specifies the probability of a neuron being dropped out.

- Dropout is typically used on hidden layers, but it can also be used on the input layer.
- You can use dropout on all hidden layers, or you can only use it on some of the layers.
- You can experiment with different dropout rates to see what works best for your model.
- Dropout is typically used during training, but you can also use it during inference if you want to make your model more robust to noise.

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."

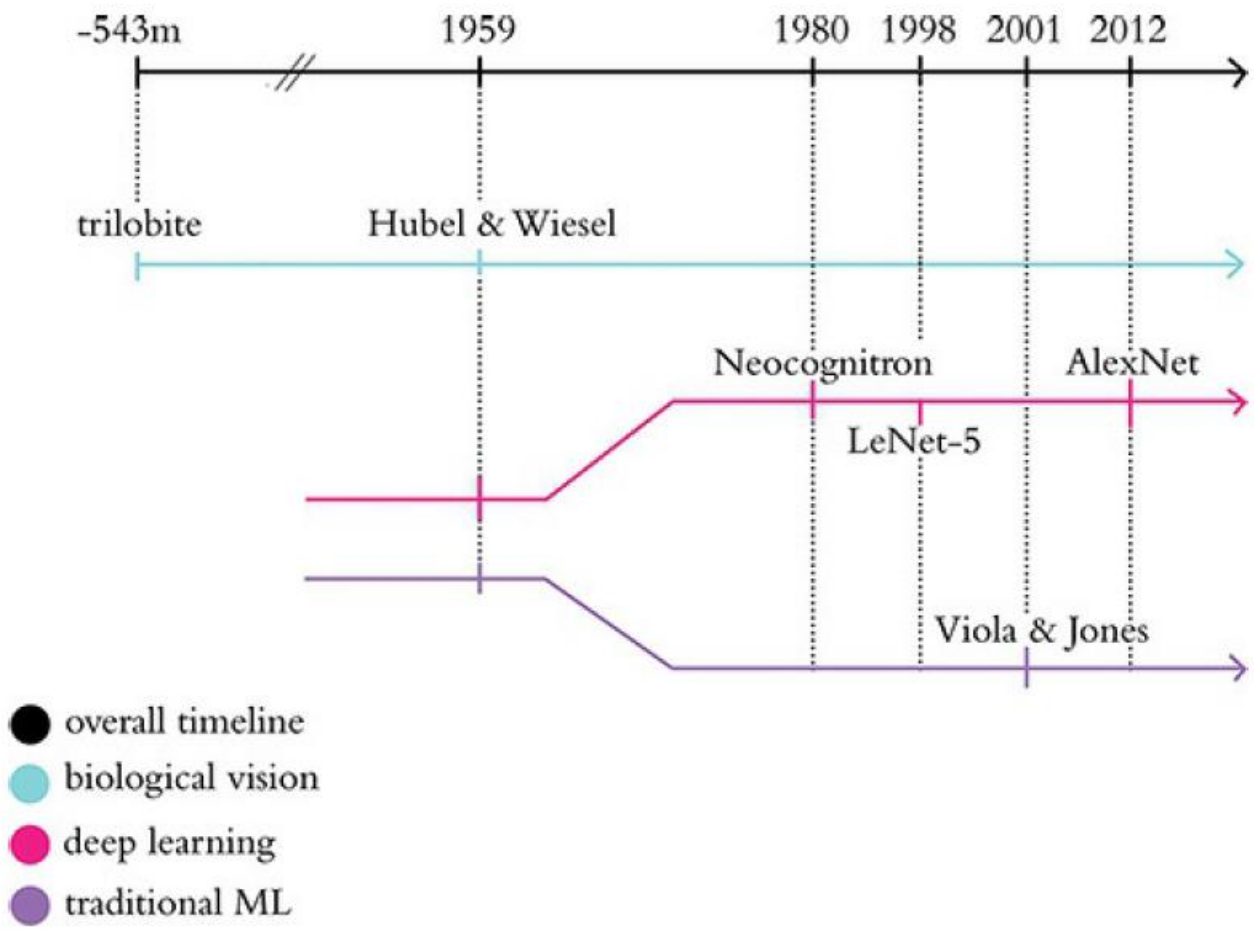


Figure 1.8 Abridged timeline of biological and machine vision, highlighting the key historical moments in the deep learning and traditional machine learning approaches to vision that are covered in this section

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

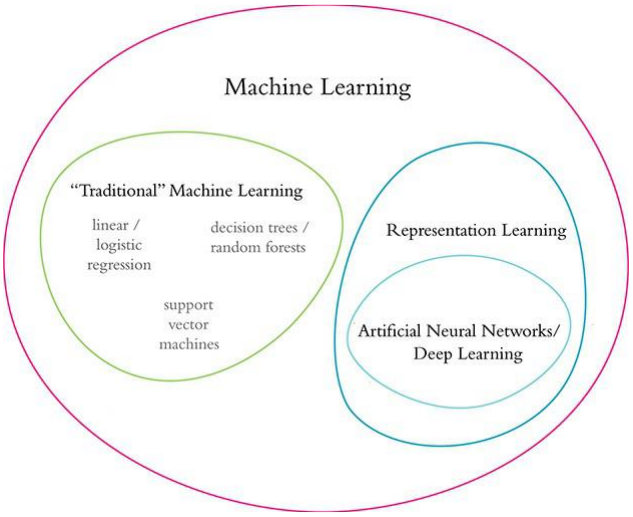
Deep learning is a type of machine learning that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain and are able to learn complex patterns from data. This makes them well-suited for natural language processing (NLP), which is the field of computer science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

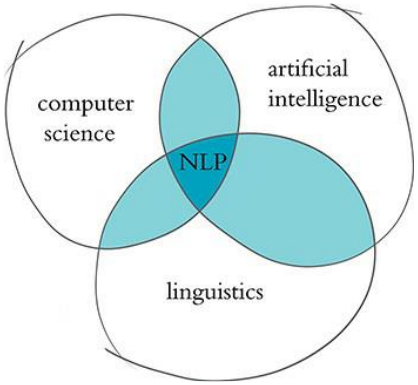
This manual creation of features is often a highly specialized task. For working with language data, for example, it might require graduate-level training in linguistics.

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.

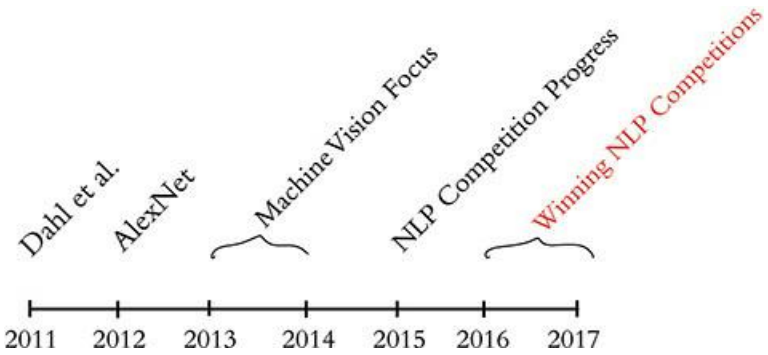
4. *Speech recognition*: interpreting voice commands to provide information or act, as with virtual assistants like Amazon's Alexa, Apple's Siri, or Microsoft's Cortana.

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, 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], 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, 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

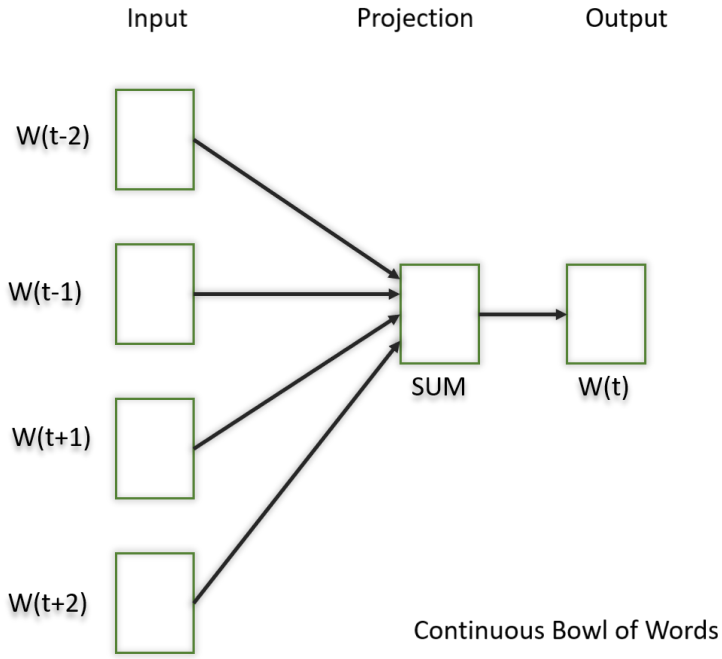
1) Word2Vec:

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:

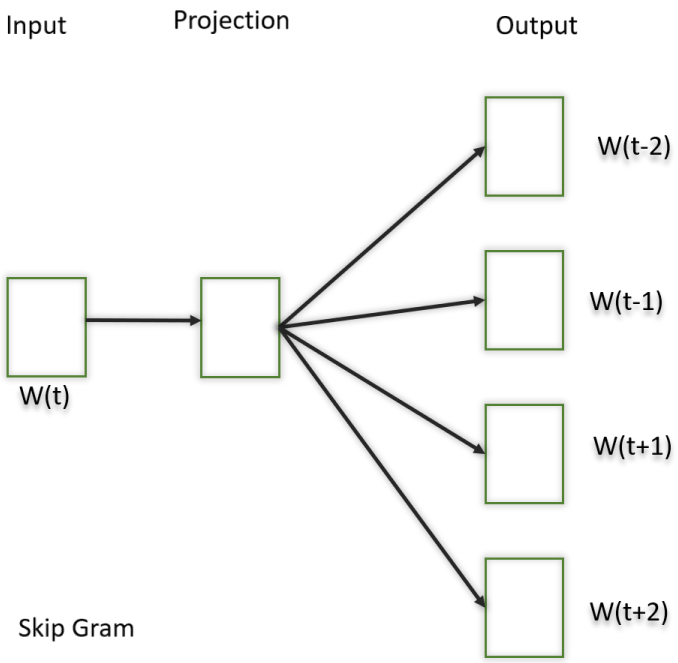
1.1) Continuous Bowl of Words(CBOW)

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.

$$V_{king} - V_{man} + V_{woman} = V_{queen}$$
$$V_{bezos} - V_{amazon} + V_{tesla} = V_{musk}$$
$$V_{windows} - V_{microsoft} + V_{google} = V_{android}$$

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.

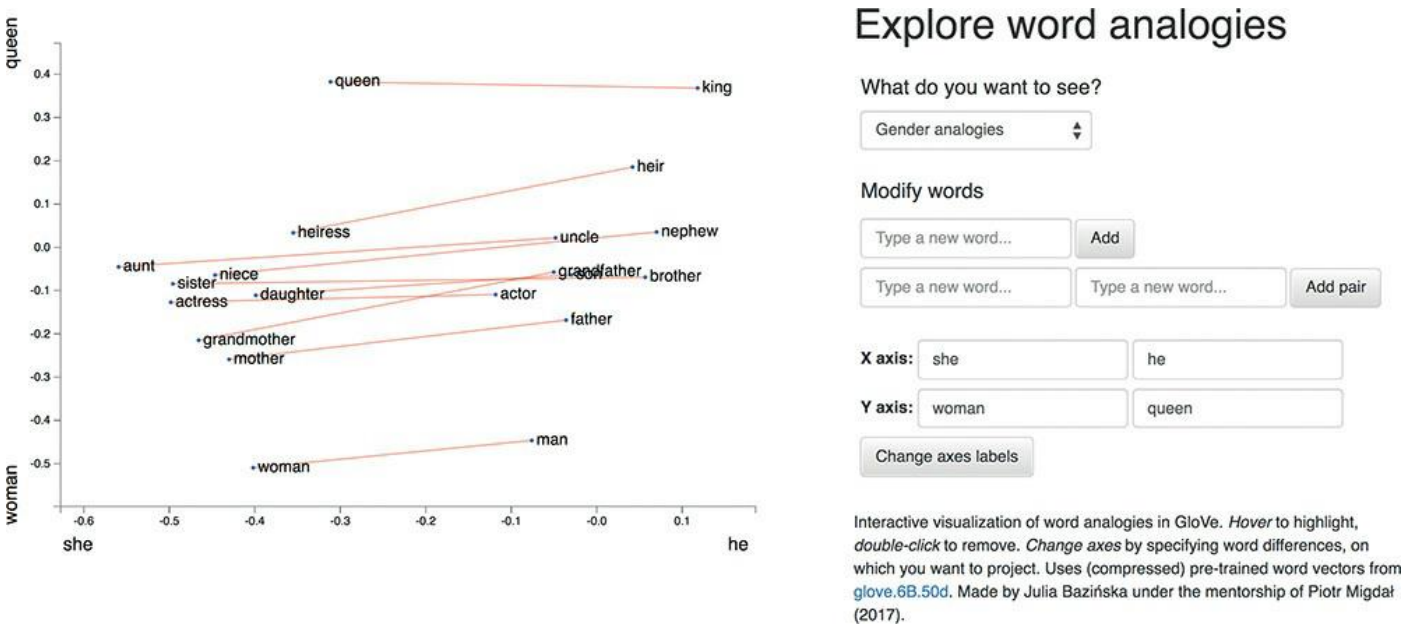
Word2viz also allows you to specify a context for the word embeddings. This can be useful for exploring the relationships between words in a specific domain or topic. For example, you could specify the context of "technology" to explore the relationships between words related to technology.

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://lamyiowce.github.io/word2viz/***



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

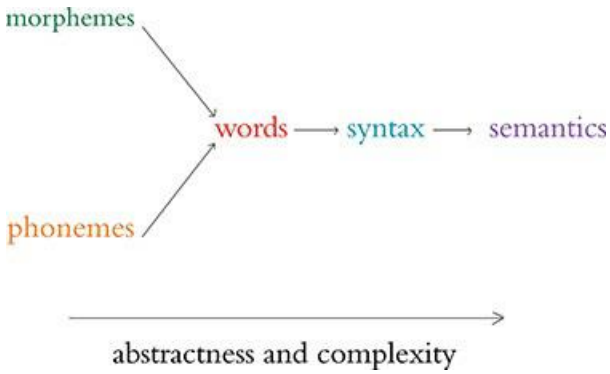
Phonology is the study of how sounds are used to make words in a language. Every language has a different set of sounds, and these sounds are combined to form words. The traditional way to study phonology is to break down speech into small pieces and label each piece with a specific sound. With deep 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 placeholding 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.

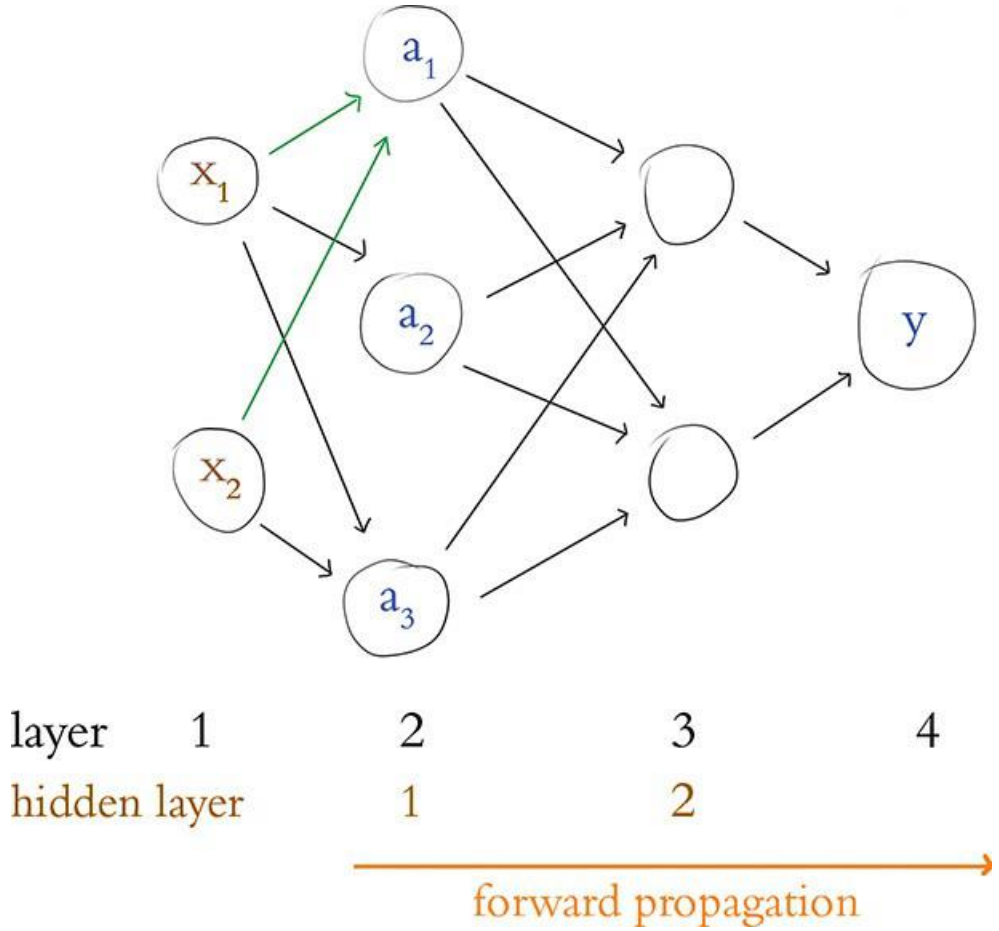
3.2.DENSE LAYERS

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 , 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 , 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.**

This particular neuron, like its siblings **a1** and **a2** , receives input regarding a given object's "ketchup-y-ness" and "mustard-y-ness" from **x1** and **x2** , 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**

$z = w \cdot x + b$

$= w1x1 + w1x2 + b$

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

$= -2 + 4.5 - 0.9$

$= 1.6$

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

$a = max(0, z)$

$= max(0, 1.6)$

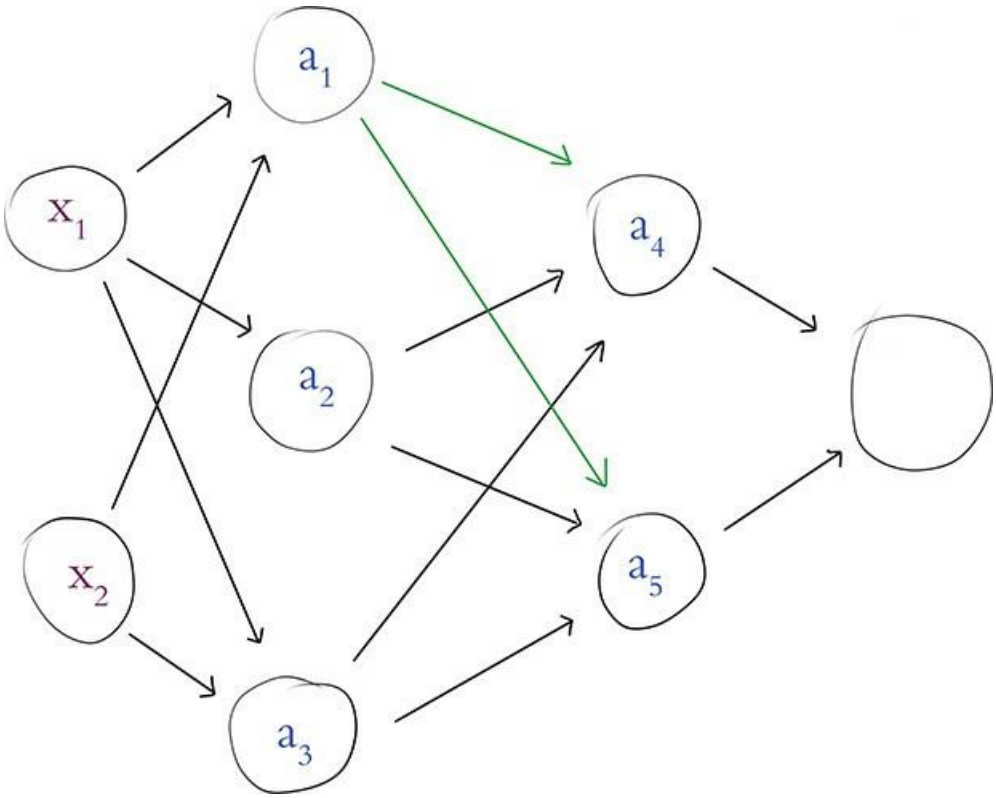
$= 1.6$

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 **a2** and **a3** —we would follow the same process as we did for the neuron labeled **a1** .

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 **a1** , the activation it outputs (**a1** = 1.6) becomes one of the three inputs into the neuron labeled **a4** (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 **a5**).

To provide an example of forward propagation through the second hidden layer, let's compute **a** for the neuron labeled **a4** .

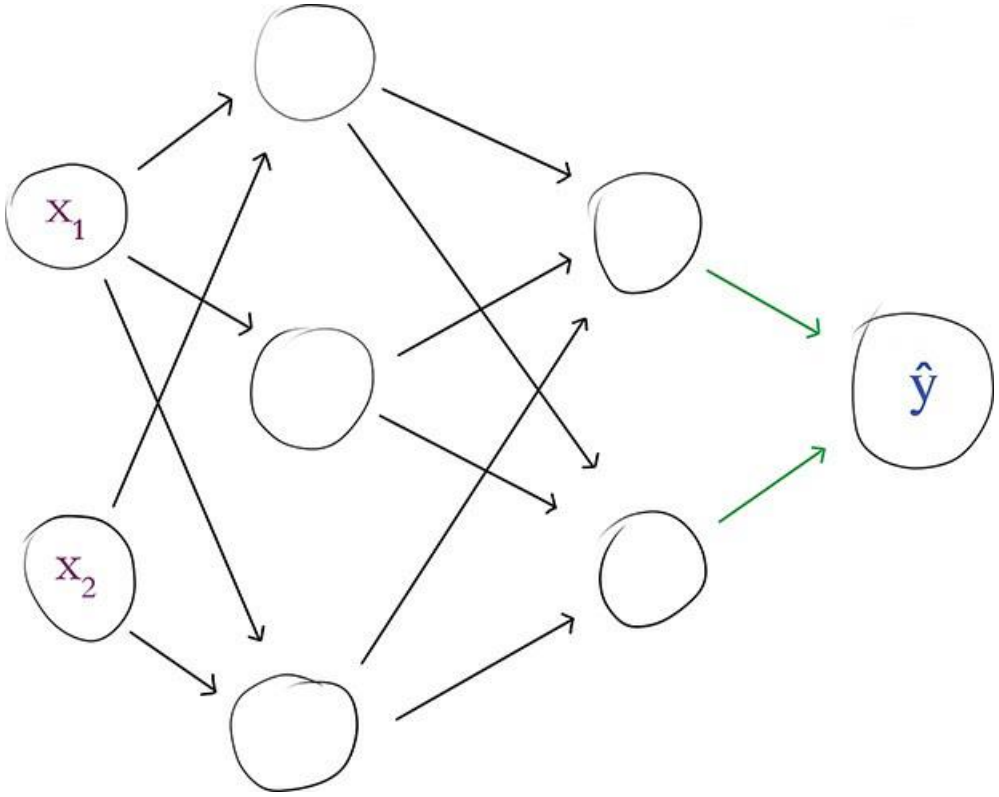
$$\begin{aligned} a &= max\,(0, z) \\ &= max\,(0, (w \cdot x + b)) \\ &= \max\,(0, (w_1 x_1 + w_2 x_2 + w_3 x_3 + b)) \end{aligned}$$

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

- **x1** is the value **a** = 1.6, which we obtained earlier from the neuron labeled **a1**

- **x2** is the activation output **a** (whatever it happens to equal) from the neuron labeled **a2**
- **x3** is likewise a unique activation **a** from the neuron labeled a3
- The unique parameters w1 , w2 , w3 , and b for this neuron would lead it to output a unique a 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.

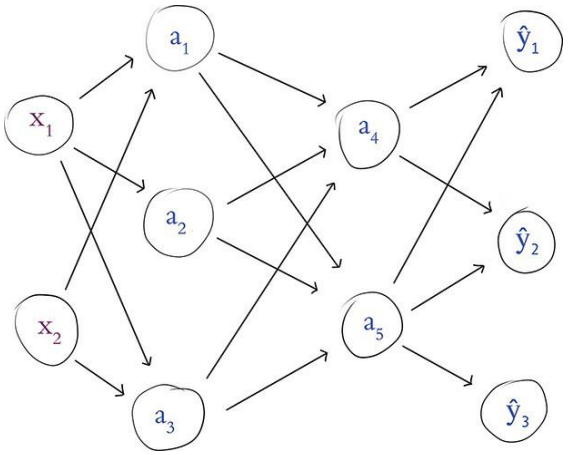


Figure: Our food-detecting network, now with three softmax neurons in the output layer

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:

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])/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])/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])/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 $n_{parameters} = n_{\mathbf{w}} + n_{\mathbf{b}}$

= 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:

MSE = (1/n) * Σ(y - ŷ)^2

Where:

- n is the number of samples in the dataset
- y is the actual value
- ŷ 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:

Binary Cross-Entropy = - (1/n) * Σ(y * log(ŷ) + (1 - y) * log(1 - ŷ))

Where:

- n is the number of samples in the dataset
- y is the actual value (0 or 1)
- ŷ 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:

Categorical Cross-Entropy = - (1/n) * ΣΣ(y(i,j) * log(ŷ(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
- ŷ(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 $\hat{\mathbf{y}}$ gradually converges toward the target of $\mathbf{y}$, 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 $\hat{\mathbf{y}}$ values.

3. A cost function (e.g., cross-entropy cost) evaluates the network's $\hat{\mathbf{y}}$ values against the true $\mathbf{y}$ values, providing a cost $\mathbf{C}$ 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

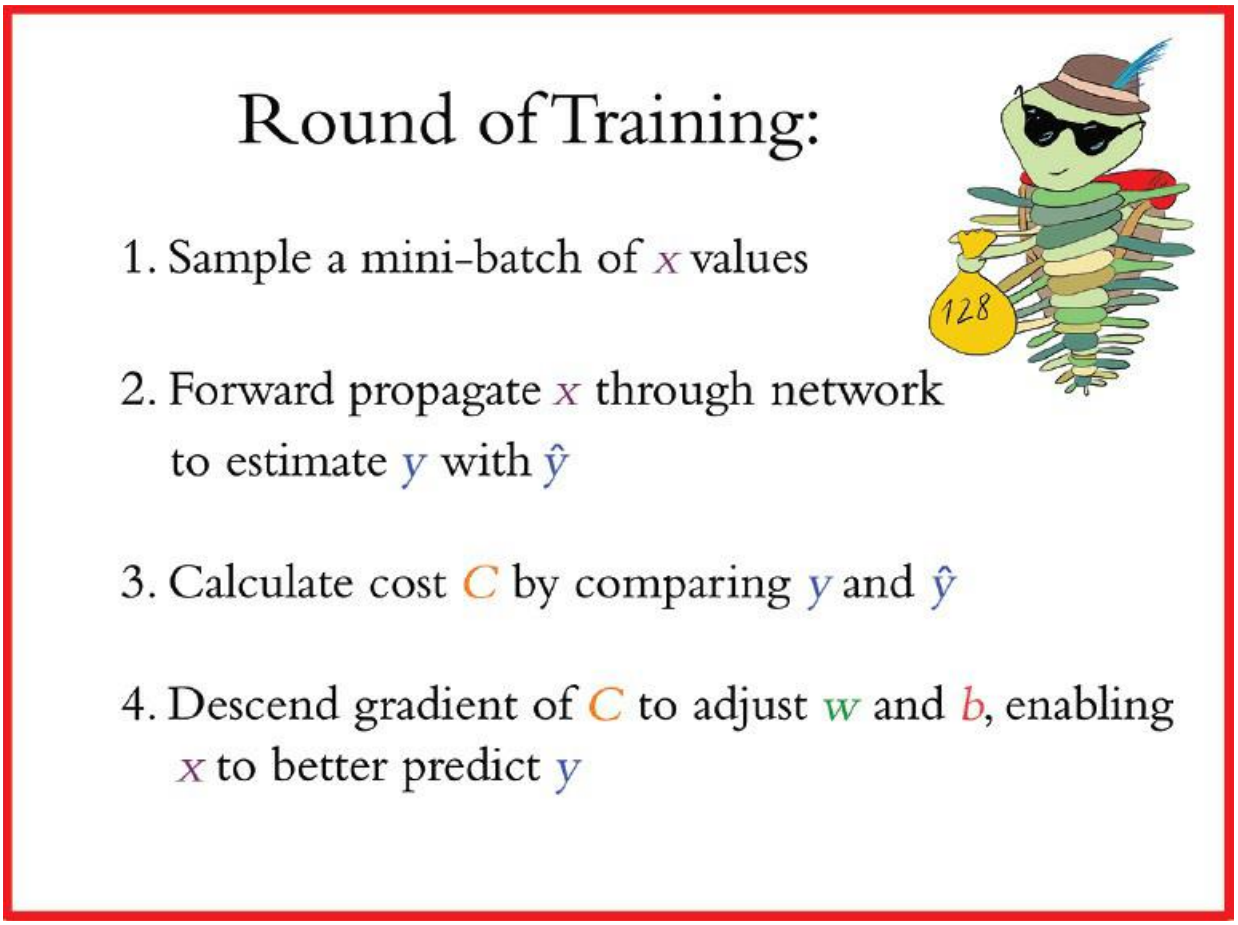


Figure 8.6 captures how rounds of training are repeated until we run out of training images to sample.

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.

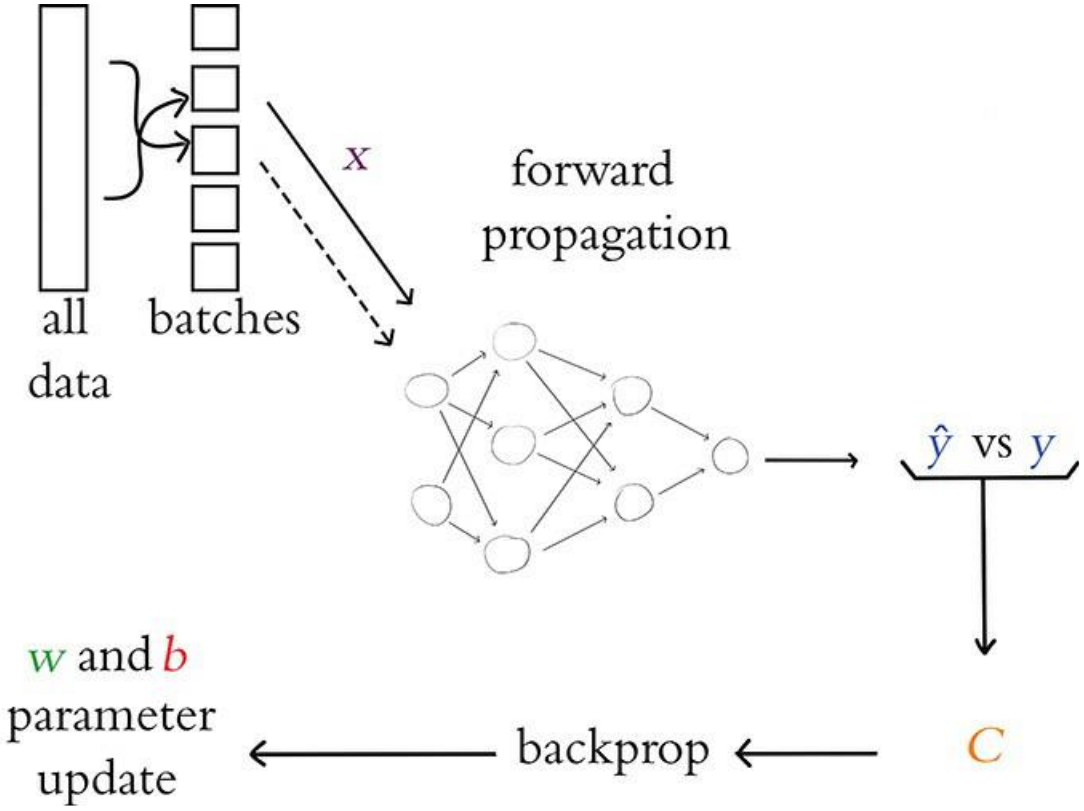


Figure 8.6 An outline of the overall process for training a neural network with stochastic gradient descent.

- 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

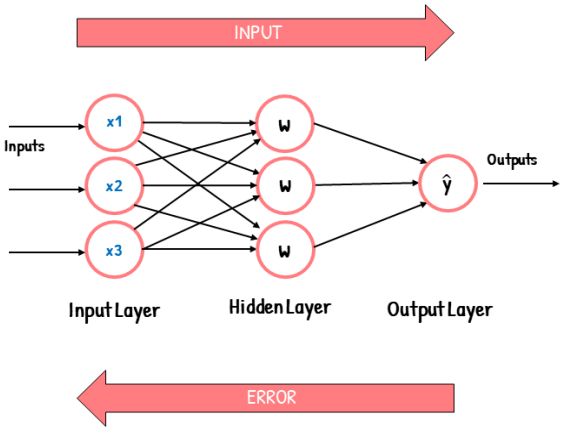
There are a number of techniques that can be used to escape the local minimum in gradient descent:

- **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?

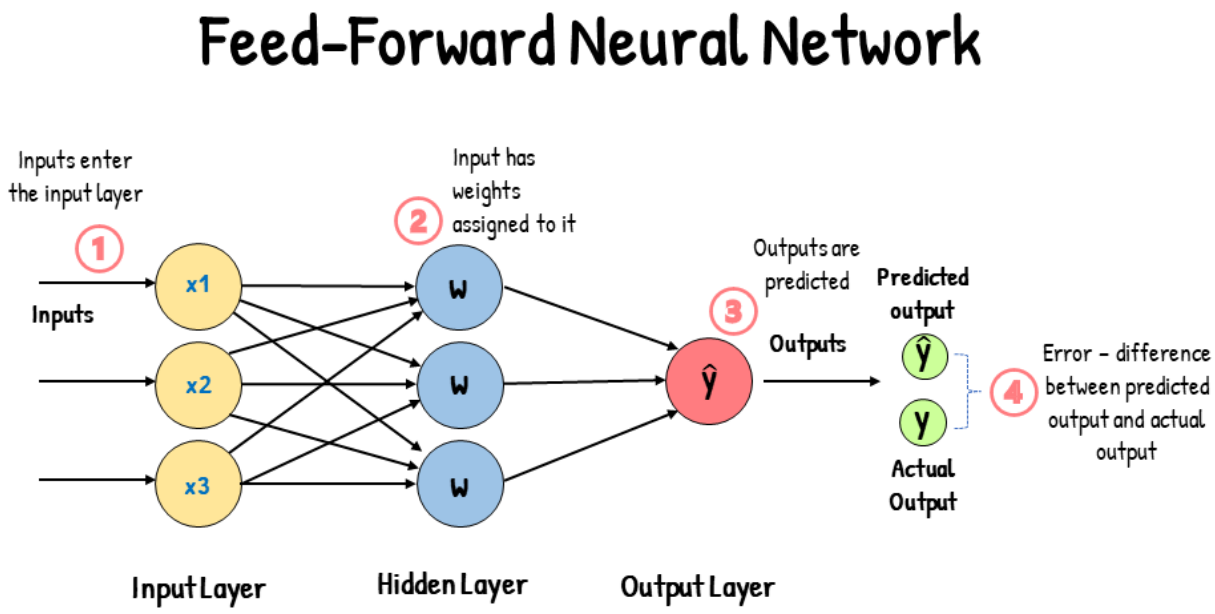
To understand how backpropagation works, let's first understand how a feedforward network works.

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.

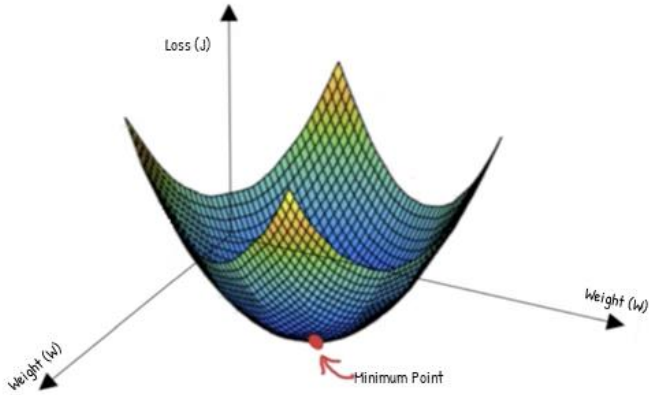
The weights associated with each input are numerical values. These weights are an indicator of the importance of the input in predicting the final output. For example, an input associated with a large weight will have a greater influence on the output than an input associated with a small weight.

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.



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.

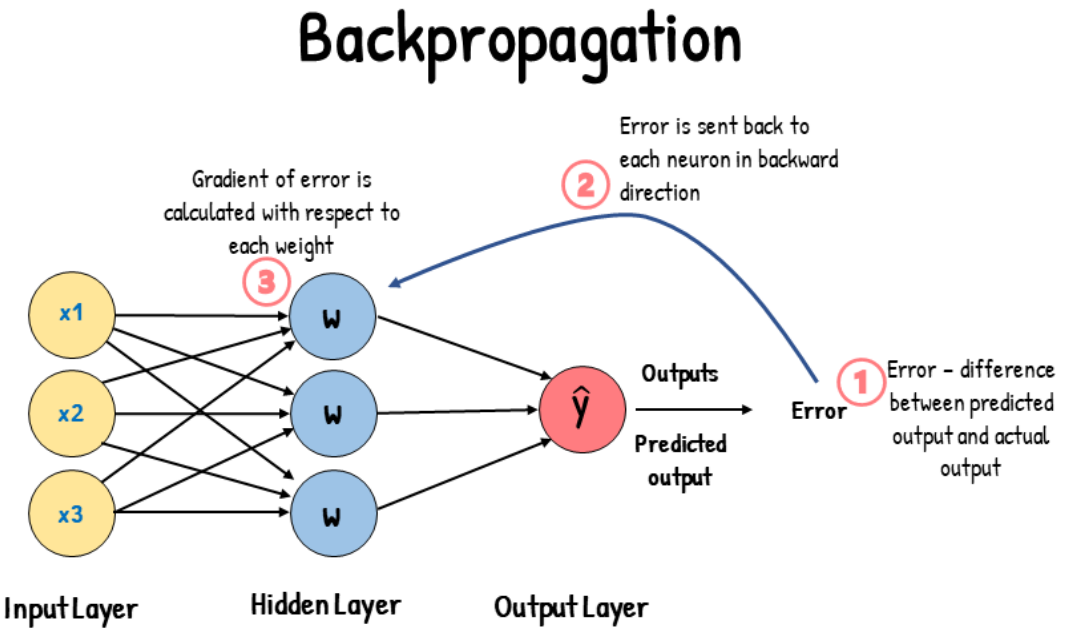
The **cost function (J) indicates how accurately the model performs**. It tells us how far-off our predicted output values are from our actual values. It is also known as the error. Because the cost function quantifies the error, we aim to minimize the cost function.



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	•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.

- 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.

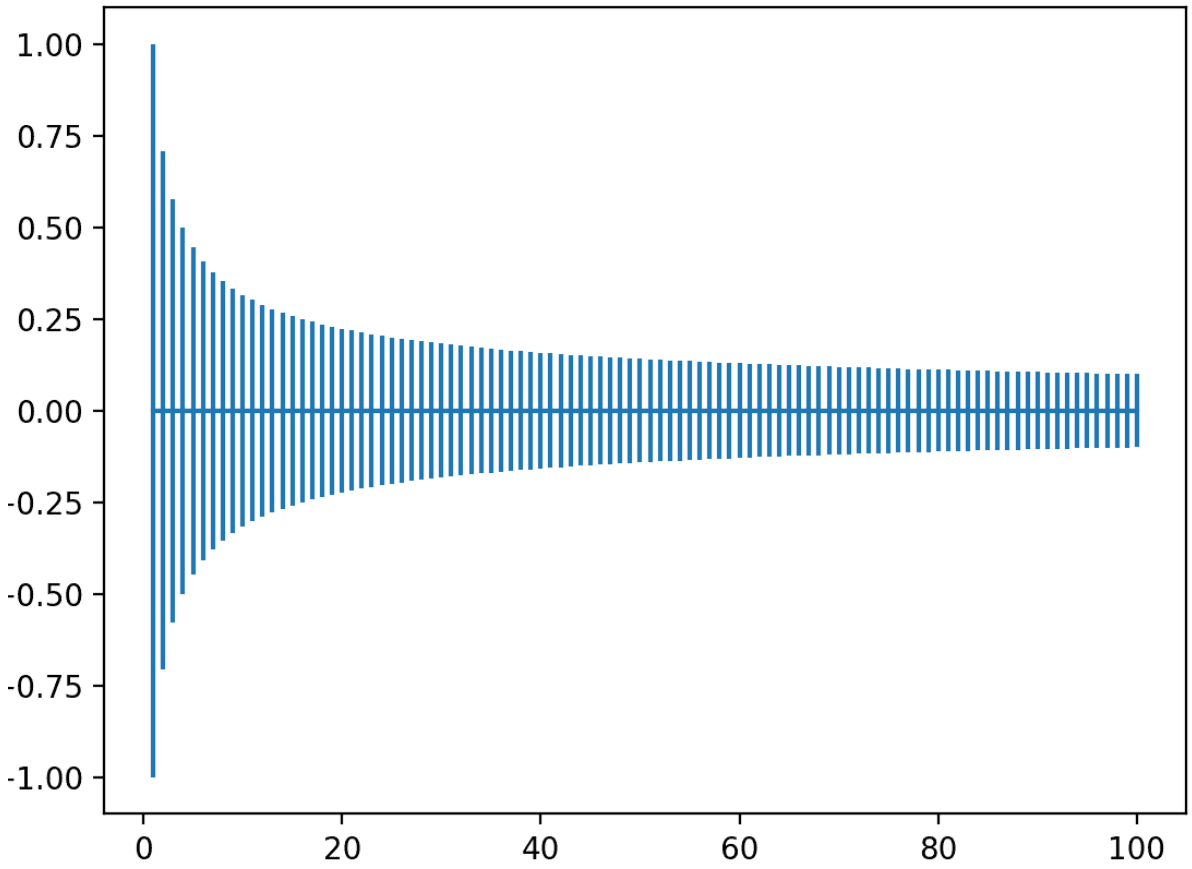
The complete example is listed below.

```
# 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 –

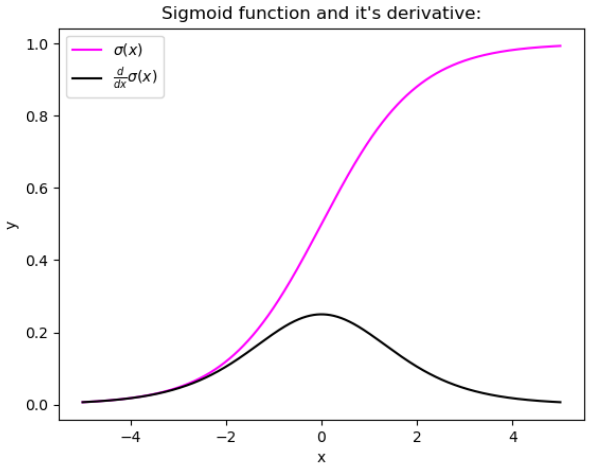
As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges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

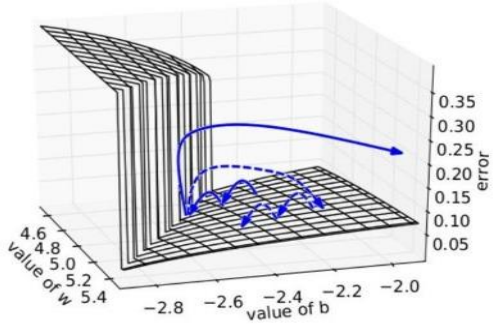
Why do the gradients even vanish/explode?

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	Vanishing
There is an exponential growth in the model parameters.	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 NaN during training.	The model weights may become 0 during training.
The model experiences avalanche learning.	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 (BatchNo (None, 784)               3136
_________________________________________________________________
dense_9 (Dense)              (None, 300)               235500
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
dense_10 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_2 (Batch (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.

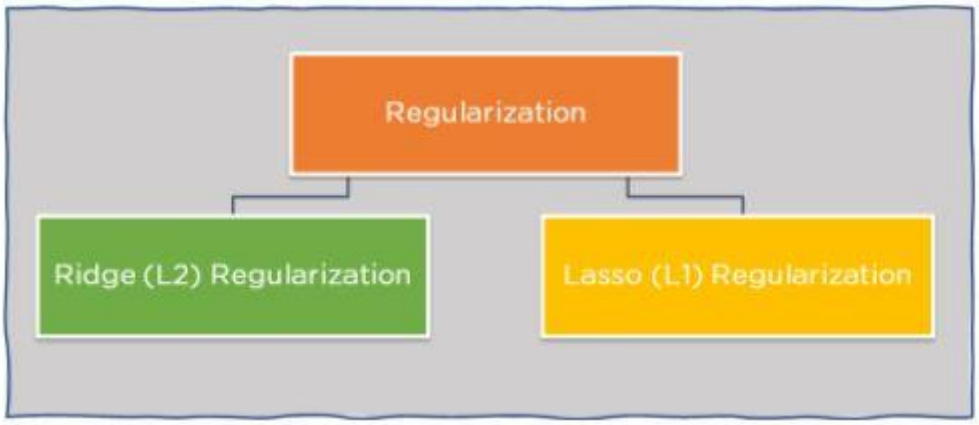


Figure 6: Regularization techniques

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:

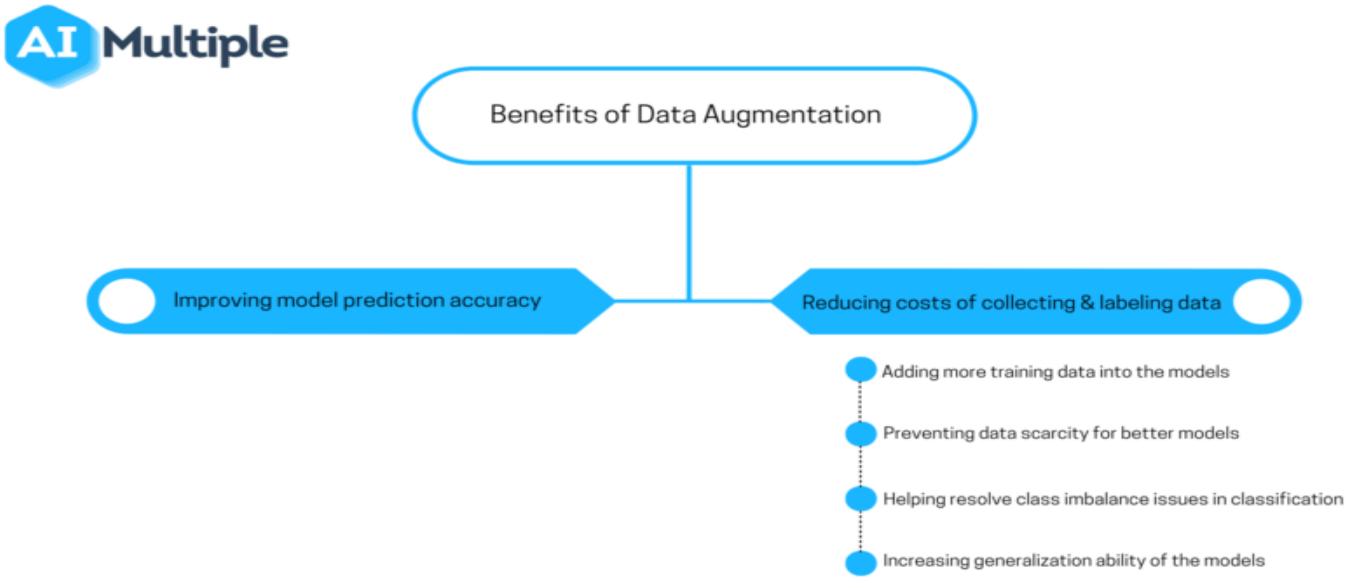
- In linear regression, regularization can be used to prevent the model from overfitting the training data. This can be done by adding a penalty to the sum of the squared values of the model weights.
- In logistic regression, regularization can be used to prevent the model from becoming too sensitive 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 p oints from existing data. This includes making small changes to data or using deep learning models to generat e 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 ∇J(θ;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**

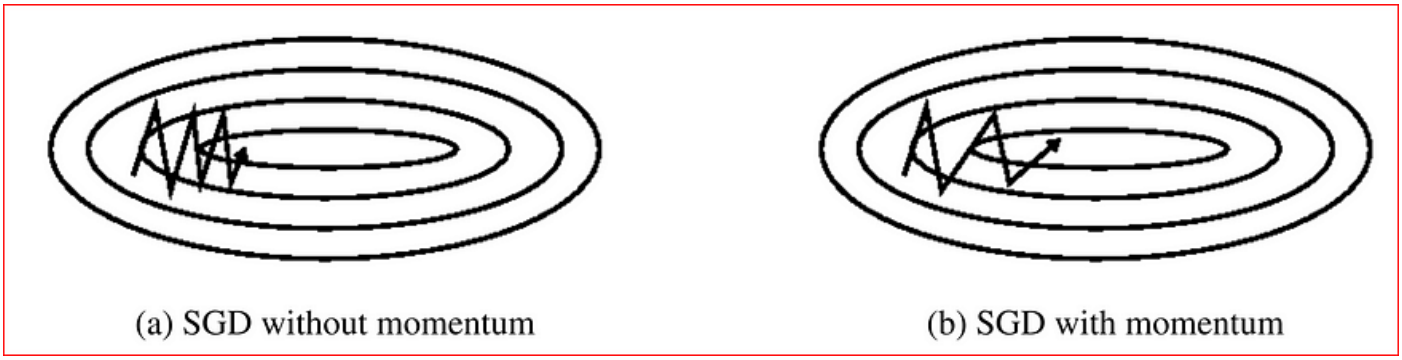
Role of an optimizer

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.



(a) SGD without momentum (b) SGD with momentum

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.

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

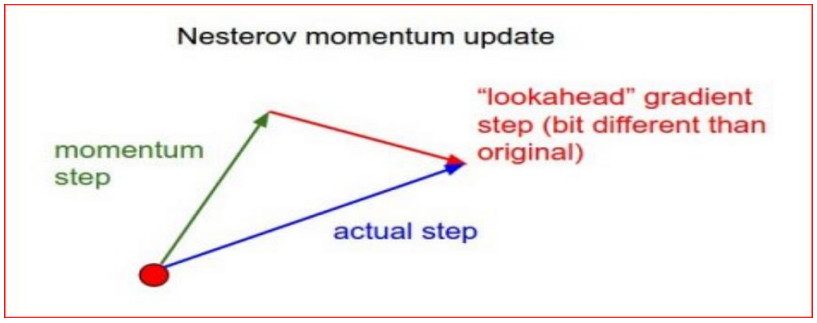
$$\theta = \theta - v_t$$

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. **Adadelta, RMSProp, and adam tries to resolve Adagrad's radically diminishing learning rates**.

4.Adadelta

- Adadelta 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 Adadelta 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

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

γ is the decay term that takes value from 0 to 1. gt 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(mt) and the squared gradient(vt) which is the estimates of the first and second moment.

Hyper-parameters β1, β2 ∈ [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

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

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))
```

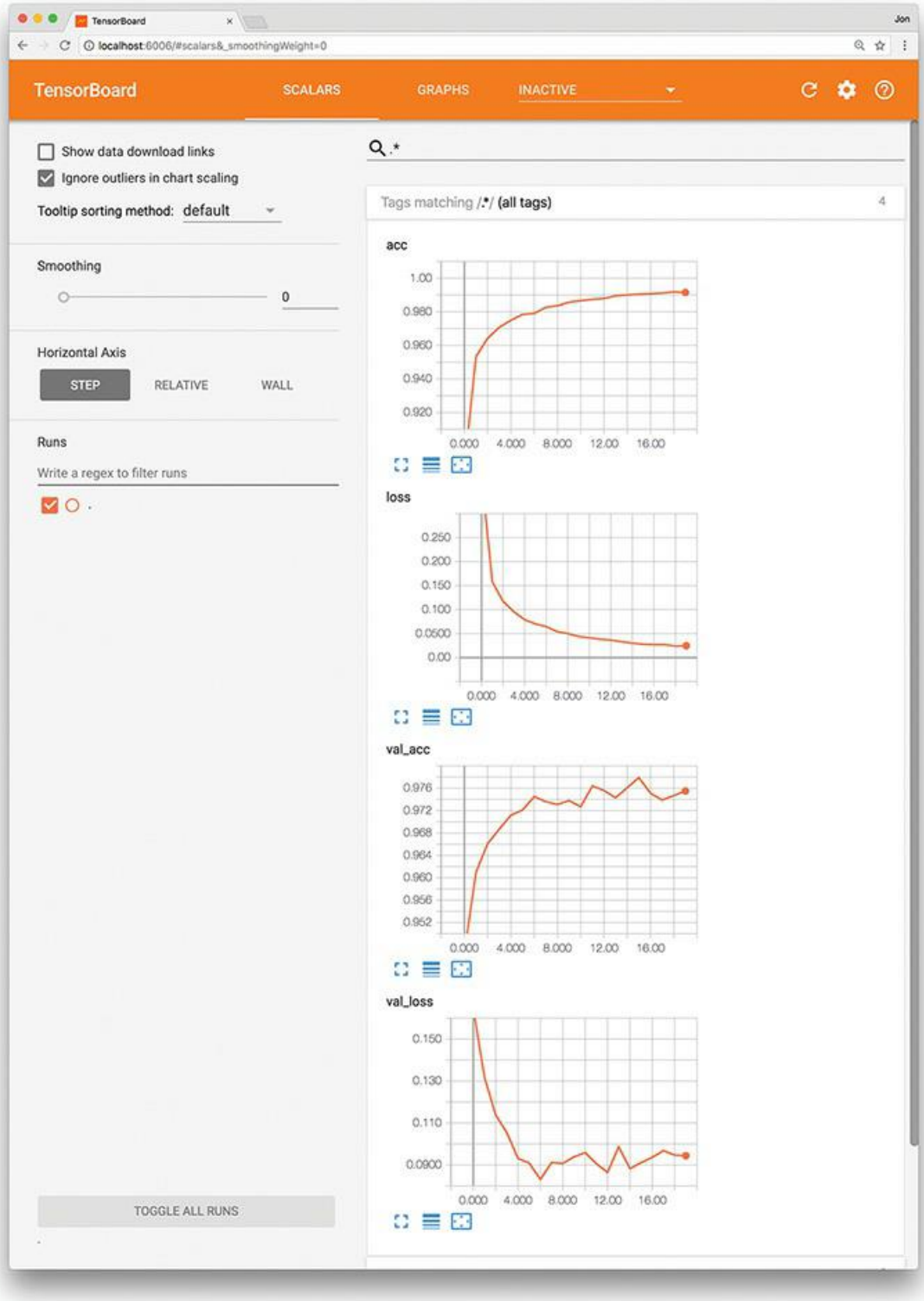
6. Predicting the median house price in a particular suburb of Boston

```
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



UNIT III: Neural Networks: Anatomy of Neural Network, Introduction to Keras: Keras, TensorFlow, Theano and CNTK, Setting up Deep Learning Workstation, Classifying Movie Reviews: Binary Classification, Classifying newswires: Multiclass Classification.

.....

3.1. Anatomy of a neural network

Training a neural network revolves around the following objects:

- *Layers*, which are combined into a *network* (or *model*)
- The *input data* and corresponding *targets*
- The *loss function*, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

You can visualize their interaction as illustrated in figure 3.1: the network, composed of layers that are chained together, maps the input data to predictions.

The loss function then compares these predictions to the targets, producing a loss value: a measure of how well the network's predictions match what was expected.

The optimizer uses this loss value to update the network's weights.

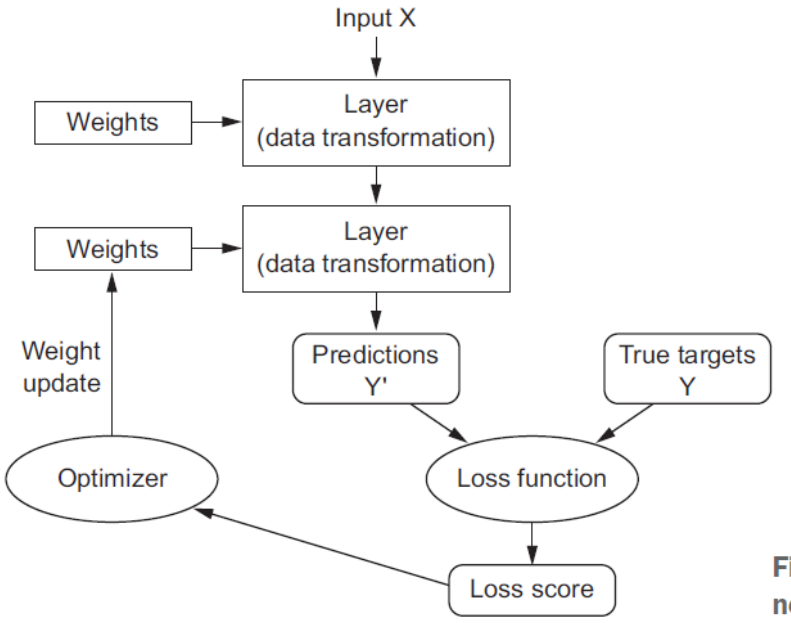


Figure 3.1 Relationship between the network, layers, loss function, and optimizer

3.1.1 Layers: the building blocks of deep learning

The fundamental data structure in neural networks is the *layer.*

A layer is a data-processing module that takes as input one or more tensors and that outputs one or more tensors. Some layers are stateless, but more frequently layers have a state.

Different layers are appropriate for different tensor formats and different types of data processing.

For instance, simple vector data, stored in 2D tensors of shape (samples, features), is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the Dense class in Keras). Sequence data, stored in 3D tensors of shape (samples, timesteps, features), is typically processed by *recurrent* layers such as an LSTM layer. Image data, stored in 4D tensors, is usually processed by 2D convolution layers (Conv2D).

Building deep-learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines.

Consider the following example

```
from keras import layers

layer = layers.Dense(32, input_shape=(784,))
```

A dense layer with 32 output units

We're creating a layer that will only accept as input 2D tensors where the first dimension is 784 (axis 0, the batch dimension, is unspecified, and thus any value would be accepted). This layer will return a tensor where the first dimension has been transformed to be 32.

When using Keras, you don't have to worry about compatibility, because the layers you add to your models are dynamically built to match the shape of the incoming layer. For instance, suppose you write the following

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(32))
```

The second layer didn't receive an input shape argument—instead, it automatically inferred its input shape as being the output shape of the layer that came before.

3.1.2 Models: networks of layers

A deep-learning model is a directed, acyclic graph of layers. The most common instance is a linear stack of layers, mapping a single input to a single output.

But as you move forward, you'll be exposed to a much broader variety of network topologies. Some common ones include the following:

- Two-branch networks
- Multihead networks
- Inception blocks

Picking the right network architecture is more an art than a science; and although there are some best practices and principles you can rely on, only practice can help you become a proper neural-network architect.

3.1.3 Loss functions and optimizers: keys to configuring the learning process

Once the network architecture is defined, you still have to choose two more things:

a) *Loss function (objective function)*—The quantity that will be minimized during training. It represents a measure of success for the task at hand.

b) *Optimizer*—Determines how the network will be updated based on the loss function.

3.2 Introduction to Keras

Keras is a deep-learning framework for Python that provides a convenient way to define and train almost any kind of deep-learning model. Keras was initially developed for researchers, with the aim of enabling fast experimentation.

Keras has the following key features:

- ❖ It allows the same code to run seamlessly on CPU or GPU.
- ❖ It has a user-friendly API that makes it easy to quickly prototype deep-learning models.

- ❖ It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- ❖ It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a generative adversarial network to a neural Turing machine.

Keras is distributed under the permissive MIT license, which means it can be freely used in commercial projects. It's compatible with any version of Python from 2.7 to 3.6 (as of mid-2017).

Keras has well over 200,000 users, ranging from academic researchers and engineers at both startups and large companies to graduate students and hobbyists.

Keras is used at Google, Netflix, Uber, CERN, Yelp, Square, and hundreds of startups working on a wide range of problems. Keras is also a popular framework on Kaggle, the machine-learning competition website, where almost every recent deep-learning competition has been won using Keras models.

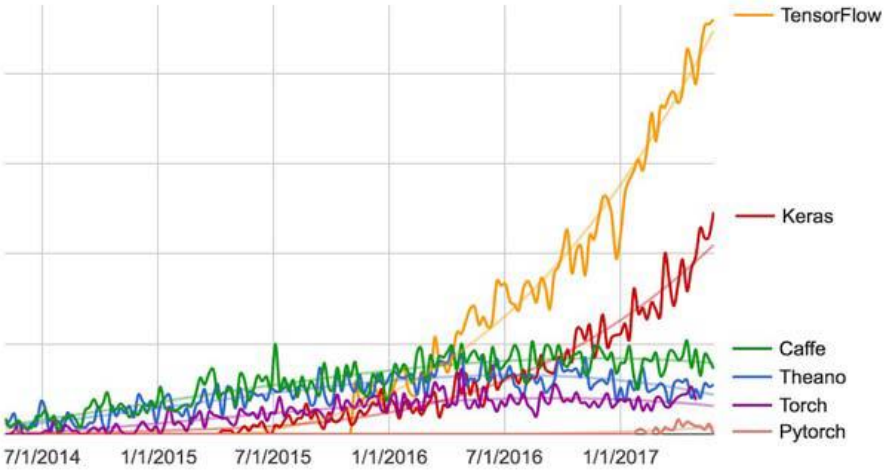


Figure: Google web search interest for different deep-learning frameworks over time

3.2.1 Keras, TensorFlow, Theano, and CNTK

Keras is a model-level library, providing high-level building blocks for developing deep-learning models.

It doesn't handle low-level operations such as tensor manipulation and differentiation. Instead, it relies on a specialized, well-optimized tensor library to do so, serving as the *backend engine* of Keras. Rather than choosing a single tensor library and tying the implementation of Keras to that library, Keras handles the problem in a modular way (see figure 3.3); thus several different backend engines can be plugged seamlessly into Keras.

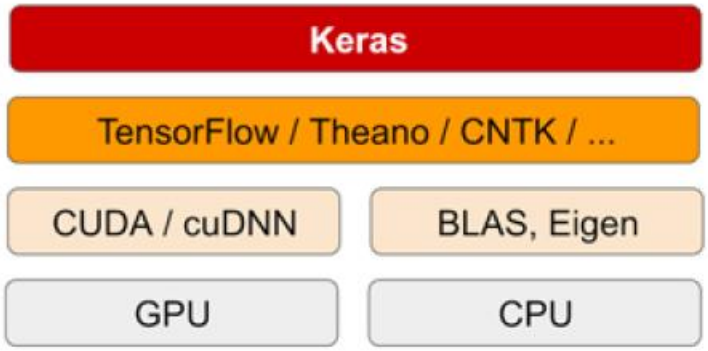


Figure 3.3 The deep-learning software and hardware stack

Currently, the three existing backend implementations are the TensorFlow backend, the Theano backend, and the Microsoft Cognitive Toolkit (CNTK) backend. In the future, it's likely that Keras will be extended to work with even more deep-learning execution engines.

- TensorFlow, CNTK, and Theano are some of the primary platforms for deep learning today.
- Theano (http://deeplearning.net/software/theano) is developed by the MILA lab at *Université de Montréal*, TensorFlow (www.tensorflow.org) is developed by Google, and
- CNTK (https://github.com/Microsoft/CNTK) is developed by Microsoft

Any piece of code that you write with Keras can be run with any of these backends without having to change anything in the code.

3.2.2 Developing with Keras: a quick overview

You've already seen one example of a Keras model: the MNIST example. The typical Keras workflow looks just like that example:

1. Define your training data: input tensors and target tensors.

2. Define a network of layers (or *model*) that maps your inputs to your targets.

3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.

4. Iterate on your training data by calling the fit() method of your model.

There are two ways to define a model: using the Sequential class (only for linear stacks of layers, which is the most common network architecture by far) or the *functional API* (for directed acyclic graphs of layers, which lets you build completely arbitrary architectures).

3.3. Setting up Deep Learning Workstation

Before you can get started developing deep-learning applications, you need to set up your workstation. It's highly recommended, although not strictly necessary, that you run deep-learning code on a modern NVIDIA GPU

If you don't want to install a GPU on your machine, you can alternatively consider running your experiments on an AWS EC2 GPU instance or on Google Cloud Platform. But note that cloud GPU instances can become expensive over time.

3.3.1 Jupyter notebooks: the preferred way to run deep-learning experiments

- Jupyter notebooks are a great way to run deep-learning experiments.
- They're widely used in the data-science and machine-learning communities.
- A *notebook* is a file generated by the Jupyter Notebook app (https://jupyter.org), which you can edit in your browser.
- It mixes the ability to execute Python code with rich text-editing capabilities for annotating what you're doing.
- A notebook also allows you to break up long experiments into smaller pieces that can be executed independently, which makes development interactive and means you don't have to rerun all of your previous code if something goes wrong late in an experiment

3.3.2 Getting Keras running: two options

To get started in practice, we recommend one of the following two options:

1.Use the official EC2 Deep Learning AMI (https://aws.amazon.com/amazonai/amis), and run Keras experiments as Jupyter notebooks on EC2. Do this if you don't already have a GPU on your local machine.

2. Install everything from scratch on a local Unix workstation. You can then run either local Jupyter notebooks or a regular Python codebase. Do this if you already have a high-end NVIDIA GPU.

3.3.3 Running deep-learning jobs in the cloud: pros and cons

There are many pros and cons to running deep-learning jobs in the cloud. Here are some of the most important ones:

Pros:

- **Scalability:** Cloud computing provides virtually unlimited scalability, so you can easily add or remove resources as needed. This is essential for deep learning, as training models can be very demanding on resources.
- **Cost-effectiveness:** Cloud computing can be very cost-effective for deep learning, especially if you only need to use the resources for a short period of time. You can also pay for the resources you use, so you're not wasting money on unused capacity.
- **Ease of use:** Cloud computing platforms make it easy to set up and run deep-learning jobs. Many platforms provide pre-configured machine learning environments, so you don't have to worry about setting up the infrastructure yourself.
- **Collaboration:** Cloud computing makes it easy to collaborate on deep learning projects. You can share data and models with other team members, and you can also run jobs on multiple machines in parallel.
- **Security:** Cloud computing providers offer a high level of security for your data. Your data is encrypted in transit and at rest, and you can control who has access to it.

Cons:

- **Latency:** There can be some latency when running deep-learning jobs in the cloud, as the data has to travel to and from the cloud servers. This can be a problem for real-time applications.
- **Vendor lock-in:** If you choose to use a particular cloud provider, you may become locked in to their platform. This can make it difficult to switch providers if you're not happy with their services.
- **Complexity:** Cloud computing can be complex, especially if you're not familiar with it. There are a lot of different services and features to choose from, and it can be difficult to know which ones are right for you.

3.3.4 What is the best GPU for deep learning?

If you're going to buy a GPU, which one should you choose? The first thing to note is that it must be an NVIDIA GPU. NVIDIA is the only graphics computing company that has invested heavily in deep learning so far, and modern deep-learning frameworks can only run on NVIDIA cards

3.4. Classifying Movie Reviews: Binary Classification

Two-class classification, or binary classification, may be the most widely applied kind of machine-learning problem. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

3.4.1 The IMDB dataset

You'll work with the IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Listing 3.1 Loading the IMDB dataset

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

The argument num_words=10000 means you'll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of manageable size

The variables train_data and test_data are lists of reviews; each review is a list of word indices (encoding a sequence of words). train_labels and test_labels are lists of 0s and 1s, where 0 stands for *negative* and 1 stands for *positive*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]

>>> train_labels[0]
1
```

Because you're restricting yourself to the top 10,000 most frequent words, no word index will exceed 10,000:

>>> max([max(sequence) for sequence in train_data])

output: 9999

For kicks, here's how you can quickly decode one of these reviews back to English words:

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

word_index is a dictionary mapping words to an integer index.

Reverses it, mapping integer indices to words

Decodes the review. Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

3.4.2 Preparing the data

You can't feed lists of integers into a neural network. You have to turn your lists into tensors. There are two ways to do that:

1. Pad your lists so that they all have the same length, turn them into an integer tensor of shape (samples, word_indices), and then use as the first layer in your network a layer capable of handling such integer tensors (the Embedding layer, which we'll cover in detail later in the book).

2. One-hot encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s. Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

Let's go with the latter solution to vectorize the data, which you'll do manually for maximum clarity

Listing 3.2 Encoding the integer sequences into a binary matrix

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Creates an all-zero matrix of shape (len(sequences), dimension)

Sets specific indices of results[i] to 1s

Vectorized training data

Vectorized test data

Here's what the samples look like now:

>>> x_train[0]

array([0., 1., 1., ..., 0., 0., 0.])

You should also vectorize your labels, which is straightforward:

y_train = np.asarray(train_labels).astype('float32')

y_test = np.asarray(test_labels).astype('float32')

3.4.3 Building your network

The input data is vectors, and the labels are scalars (1s and 0s): this is the easiest setup you'll ever encounter. A type of network that performs well on such a problem is a simple stack of fully connected (Dense) layers with relu activations: Dense(16, activation='relu').

Listing 3.3 The model definition

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

What are activation functions, and why are they necessary?

Without an activation function like `relu` (also called a *non-linearity*), the `Dense` layer would consist of two linear operations—a dot product and an addition:

```
output = dot(W, input) + b
```

So the layer could only learn *linear transformations* (affine transformations) of the input data: the *hypothesis space* of the layer would be the set of all possible linear transformations of the input data into a 16-dimensional space. Such a hypothesis space is too restricted and wouldn't benefit from multiple layers of representations, because a deep stack of linear layers would still implement a linear operation: adding more layers wouldn't extend the hypothesis space.

In order to get access to a much richer hypothesis space that would benefit from deep representations, you need a non-linearity, or activation function. `relu` is the most popular activation function in deep learning, but there are many other candidates, which all come with similarly strange names: `prelu`, `elu`, and so on.

Finally, you need to choose a loss function and an optimizer. Because you're facing a binary classification problem and the output of your network is a probability (you end your network with a single-unit layer with a sigmoid activation), it's best to use the binary_crossentropy loss. It isn't the only viable choice: you could use, for instance, mean_squared_error. But crossentropy is usually the best choice when you're dealing with models that output probabilities.

Listing 3.4 Compiling the model

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

3.4.4 Validating your approach

In order to monitor during training the accuracy of the model on data it has never seen before, you'll create a validation set by setting apart 10,000 samples from the original training data

```
In [14]: # Input for Validation
         X_val = X_train[:10000]
         partial_X_train = X_train[10000:]

         # Labels for validation
         y_val = y_train[:10000]
         partial_y_train = y_train[10000:]
```

You'll now train the model for 20 epochs (20 iterations over all samples in the x_train and y_train tensors), in mini-batches of 512 samples. At the same time, you'll monitor loss and accuracy on the 10,000 samples that you set apart. You do so by passing the validation data as the validation_data argument.

```
In [15]: history = model.fit(partial_X_train,
                             partial_y_train,
                             epochs=20,
                             batch_size=512,
                             validation_data=(X_val, y_val))
```

let's use Matplotlib to plot the training and validation loss side by side (see figure 3.7), as well as the training and validation accuracy (see figure 3.8).

Listing 3.9 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

"bo" is for "blue dot."

"b" is for "solid blue line."

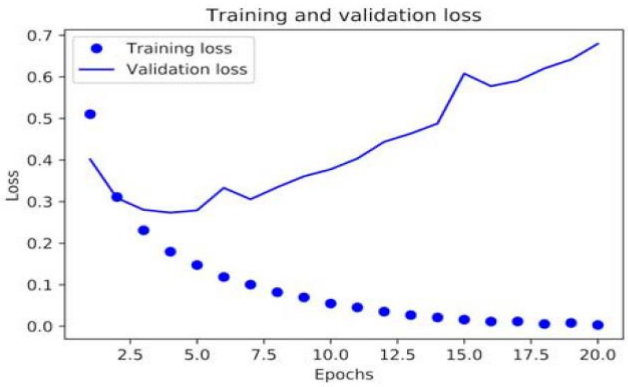


Figure 3.7 Training and validation loss

As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. That's what you would expect when running gradientdescent optimization—the quantity you're trying to minimize should be less with every iteration.

3.5. Classifying newswires: Multiclass Classification

In this example, we will build a model to classify Reuters newswires into 46 mutually exclusive topics. Because we have many classes, this problem is an instance of multiclass classification; and because each data point should be classified into only one category, the problem is more specifically an instance of single-label, multiclass classification. If each data point could belong to multiple categories (in this case, topics), you'd be facing a multilabel, multiclass classification problem.

Reuters dataset

We will work with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It's a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set. Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras.

In [1]:
```
import numpy as np
import pandas as pd
import warnings
import tensorflow as tf # import tensor flow
import numpy as np
import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
```

2.

Load Dataset

In [2]:
```
from tensorflow.keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)
```

3.

In [3]:
```
len(train_data),len(test_data)
```

Out[3]:
```
(8982, 2246)
```

The argument num_words=10000 restricts the data to the 10,000 most frequently occurring words found in the data. You have 8,982 training examples and 2,246 test examples:

In [4]:
```
print(train_data[10])
```

```
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979, 3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 4
51, 4329, 17, 12]
```

4. **Data Prep**

vectorize the input data

```
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
x_train = vectorize_sequences(train_data)#1
x_test = vectorize_sequences(test_data)#2
```

1. Verctorize training data
2. Vencotrize testing data

vectorize the label with the exact same code as in the previous example.

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results
one_hot_train_labels = to_one_hot(train_labels)#1
one_hot_test_labels = to_one_hot(test_labels)#2
```

1. Verctorize training labels
2. Vencotrize testing labels

Note that there is a built-in way to do this in Keras:

```
from tensorflow.keras.utils import to_categorical
one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

5. **Building the model**

This topic-classification problem looks similar to the previous movie-review classificationq: in both cases, we are trying to classify short snippets of text. But there is a new constraint here: the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger.

In a stack of Dense layers like that we have been using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an information bottleneck. In the previous example, we used 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information. For this reason we will use larger layers. Let's go with 64 units.

Model Defination

```
model = keras.Sequential([
  layers.Dense(64, activation='relu'),
  layers.Dense(64, activation='relu'),
  layers.Dense(46, activation='softmax')
])
```

Note about this architecture:

1. We end the model with a Dense layer of size 46. This means for each input sample, the network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
2. The last layer uses a softmax activation. You saw this pattern in the MNIST example. It means the model will output a probability distribution over the 46 different output classes — for every input sample, the model will produce a 46-dimensional output vector, where output[i] is the probability that the sample belongs to class i. The 46 scores will sum to 1.
3. The best loss function to use in this case is categorical_crossentropy. It measures the distance between two probability distributions: here, between the probability distribution output by the model and the true distribution of the labels. By minimizing the distance between these two distributions, you train the model to output something as close as possible to the true labels.

6. Compile the model

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

Validation of the approach

Let's set apart 1,000 samples in the training data to use as a validation set.

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

let's train the model for 20 epochs.

Training the model

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Epoch 1/20

16/16 [==========] - 2s 81ms/step - loss: 3.1029 - accuracy: 0.4079 - val_loss: 1.7132 - val_accuracy: 0.6440

Epoch 2/20

16/16 [==========] - 1s 38ms/step - loss: 1.4807 - accuracy: 0.6992 - val_loss: 1.2964 - val_accuracy: 0.7230

Epoch 3/20

16/16 [==========] - 1s 36ms/step - loss: 1.0763 - accuracy: 0.7762 - val_loss: 1.1460 - val_accuracy: 0.7380

Epoch 4/20

16/16 [==========] - 1s 36ms/step - loss: 0.8441 - accuracy: 0.8245 - val_loss: 1.0389 - val_accuracy: 0.7810

Epoch 5/20

16/16 [==========] - 1s 37ms/step - loss: 0.6595 - accuracy: 0.8658 - val_loss: 0.9456 - val_accuracy: 0.8050

Epoch 6/20

16/16 [==========] - 1s 37ms/step - loss: 0.5237 - accuracy: 0.8945 - val_loss: 0.9203 - val_accuracy: 0.8040

Epoch 7/20

16/16 [==========] - 1s 36ms/step - loss: 0.4181 - accuracy: 0.9160 - val_loss: 0.8765 - val_accuracy: 0.8140

Epoch 8/20

16/16 [==========] - 1s 35ms/step - loss: 0.3485 - accuracy: 0.9316 - val_loss: 0.8895 - val_accuracy: 0.8060

Epoch 9/20

16/16 [==========] - 1s 36ms/step - loss: 0.2829 - accuracy: 0.9390 - val_loss: 0.8829 - val_accuracy: 0.8110

Epoch 10/20

16/16 [==========] - 1s 36ms/step - loss: 0.2246 - accuracy: 0.9479 - val_loss: 0.9112 - val_accuracy: 0.8140

Epoch 11/20

16/16 [==========] - 1s 36ms/step - loss: 0.1894 - accuracy: 0.9532 - val_loss: 0.9060 - val_accuracy: 0.8120

Epoch 12/20

16/16 [==========] - 1s 37ms/step - loss: 0.1765 - accuracy: 0.9538 - val_loss: 0.9068 - val_accuracy: 0.8160

Epoch 13/20

16/16 [==========] - 1s 37ms/step - loss: 0.1610 - accuracy: 0.9529 - val_loss: 0.9394 - val_accuracy: 0.8100

Epoch 14/20

16/16 [==========] - 1s 37ms/step - loss: 0.1438 - accuracy: 0.9574 - val_loss: 0.9254 - val_accuracy: 0.8190

Epoch 15/20

16/16 [==========] - 1s 35ms/step - loss: 0.1305 - accuracy: 0.9584 - val_loss: 0.9666 - val_accuracy: 0.8060

Epoch 16/20

16/16 [==========] - 1s 37ms/step - loss: 0.1291 - accuracy: 0.9562 - val_loss: 0.9537 - val_accuracy: 0.8120

Epoch 17/20

16/16 [==========] - 1s 36ms/step - loss: 0.1140 - accuracy: 0.9593 - val_loss: 1.0202 - val_accuracy: 0.8020

Epoch 18/20

16/16 [==========] - 1s 38ms/step - loss: 0.1167 - accuracy: 0.9567 - val_loss: 0.9942 - val_accuracy: 0.8070

Epoch 19/20

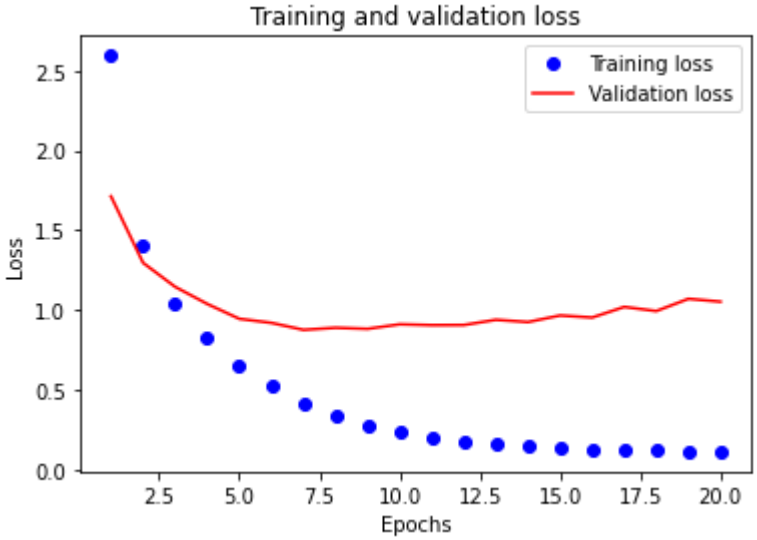
16/16 [==========] - 1s 38ms/step - loss: 0.0972 - accuracy: 0.9669 - val_loss: 1.0709 - val_accuracy: 0.7960

Epoch 20/20

16/16 [==========] - 1s 34ms/step - loss: 0.1035 - accuracy: 0.9607 - val_loss: 1.0530 - val_accuracy: 0.8020

7. Plotting the training and validation loss

```
loss = history.history['loss']

val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')

plt.plot(epochs, val_loss, 'r', label='Validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()
```


Training and validation loss
Training loss
Validation loss
Loss
Epochs
0.0
0.5
1.0
1.5
2.0
2.5
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

UNIT IV

Convolutional Neural Networks: Nerual Network and Representation Learing, Convolutional Layers, Multichannel Convolution Operation, **Recurrent Neural Networks:** Introduction to RNN, RNN Code, PyTorch Tensors: Deep Learning with PyTorch, CNN in PyTorch

……

Convolutional neural networks (CNNs) are a type of deep learning neural network that is specifically designed for image processing and computer vision tasks. CNNs are inspired by the structure and function of the human visual cortex, which is the part of the brain that is responsible for processing visual information.

CNNs have a number of advantages over other types of neural networks for image processing and computer vision tasks:

- **They are able to extract features from images that are invariant to translation, rotation, and scaling.** This means that the same features can be detected even if the object in the image is in a different position or orientation.
- **They are able to extract features from images that are hierarchically organized.** This means that they can start by extracting simple features, such as edges and corners, and then use those features to extract more complex features, such as faces and objects.
- **They are computationally efficient.** This is because the convolution operation at the heart of CNNs can be implemented using the fast Fourier transform (FFT).

CNNs have revolutionized the field of computer vision. They are now used in a wide range of applications, including:

- **Image classification:** CNNs can be used to classify images into different categories, such as cats, dogs, and cars.
- **Object detection:** CNNs can be used to detect objects in images, such as pedestrians, cars, and traffic signs.
- **Facial recognition:** CNNs can be used to recognize faces in images.
- **Medical imaging:** CNNs can be used to analyze medical images, such as X-rays and MRI scans, to diagnose diseases and identify abnormalities.
- **Natural language processing:** CNNs can be used to extract features from text, which can then be used for tasks such as sentiment analysis and machine translation.

CNNs are a powerful tool for a wide range of applications. They are still under active development, and new applications for CNNs are being discovered all the time.

Here are some specific examples of how CNNs are being used today:

- Facebook uses CNNs to recognize faces in photos.
- Google uses CNNs to power its image search engine.
- Tesla uses CNNs to power its self-driving cars.
- Doctors use CNNs to analyze medical images and diagnose diseases.
- Researchers are using CNNs to develop new methods for machine translation and text analysis.

CNNs are a powerful and versatile tool that is transforming the way we interact with the world around us.

Artificial neural networks (ANNs) Vs convolutional neural networks (CNNs)

Artificial neural networks (ANNs) and convolutional neural networks (CNNs) are both types of deep learning neural networks. However, they have different architectures and are used for different types of tasks.

ANNs are general-purpose neural networks that can be used for a variety of tasks, including classification, regression, and clustering. They are typically made up of a series of fully connected layers, meaning that each neuron in one layer is connected to every neuron in the next layer.

CNNs are a type of ANN that is specifically designed for image processing and computer vision tasks. They are made up of a series of convolutional layers, which are able to extract features from images that are invariant to translation, rotation, and scaling.

Here is a table that summarizes the key differences between ANNs and CNNs:

Characteristic	ANN	CNN
Architecture	Fully connected layers	Convolutional layers
Applications	General-purpose	Image processing, computer vision
Advantages	Flexible and versatile	Able to extract invariant features from images
Disadvantages	Can be computationally expensive	Requires a large amount of labeled training data

Which type of neural network to use depends on the specific task at hand. If you are working on a general-purpose task, such as classification or regression, then an ANN may be a good choice. If you are working on an image processing or computer vision task, then a CNN is likely to be a better choice.

Here are some examples of when to use ANNs and CNNs:

- **ANNs:**
    - Classifying text documents into different categories
    - Predicting customer churn
    - Recommending products to customers
- **CNNs:**
    - Classifying images of objects
    - Detecting objects in images
    - Segmenting images

1. Nerual Network and Representation Learing

- Neural networks initially receive data on observations, with each observation represented by some number n features.
- A simple neural network model with one hidden layer performed better than a model without that hidden layer.
- One reason is that the neural network could learn nonlinear relationships between input and output.
- However, a more general reason is that in machine learning, we often need linear combinations of our original features in order to effectively predict our target.
- Let's say that the pixel values for an MNIST digit are x1 through x784.
- There may be many other such combinations, all of which contribute positively or negatively to the probability that an image is of a particular digit.
- Neural networks can automatically discover combinations of the original features that are important through their training process.
- This process of learning which combinations of features are important is known as representation learning, and it's the main reason why neural networks are successful across different domains.

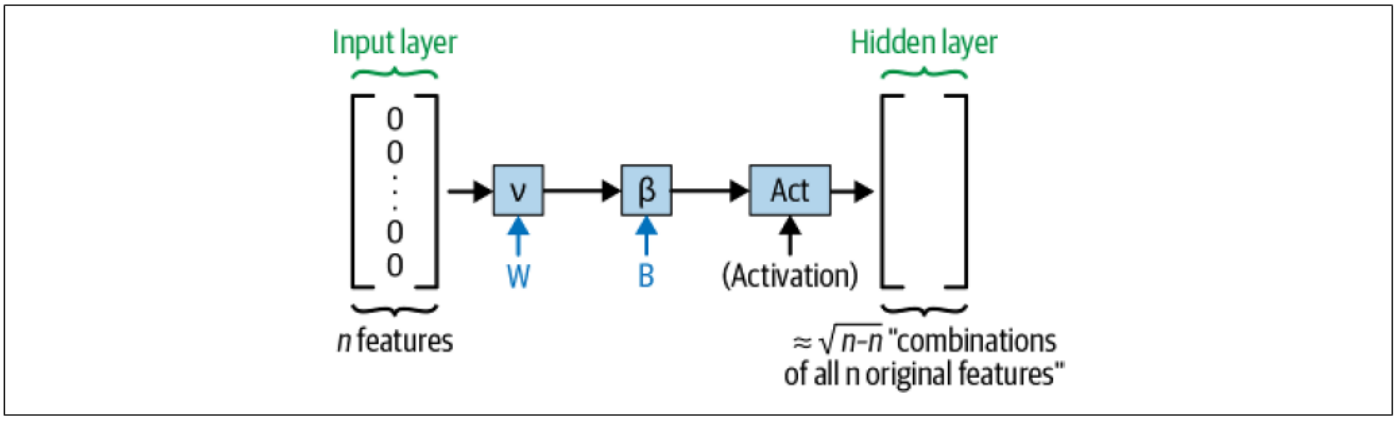


Figure 5-1. The neural networks we have seen so far start with n features and then learn somewhere between $\sqrt{n}$ and n "combinations" of these features to make predictions

Is there any reason to modify this process for image data?

The fundamental insight that suggests the answer is "yes" is that in images, the interesting "combinations of features" (pixels) tend to come from pixels that are close together in the image.

We want to exploit this fundamental fact about image data: that the order of the features matters since it tells us which pixels are near each other spatially.

But how do we do it?

1.1. A Different Architecture for Image Data

The solution, at a high level, will be to create combinations of features, as before, but an order of magnitude more of them, and have each one be only a combination of the pixels from a small rectangular patch in the input image. Figure 5-2 describes this.

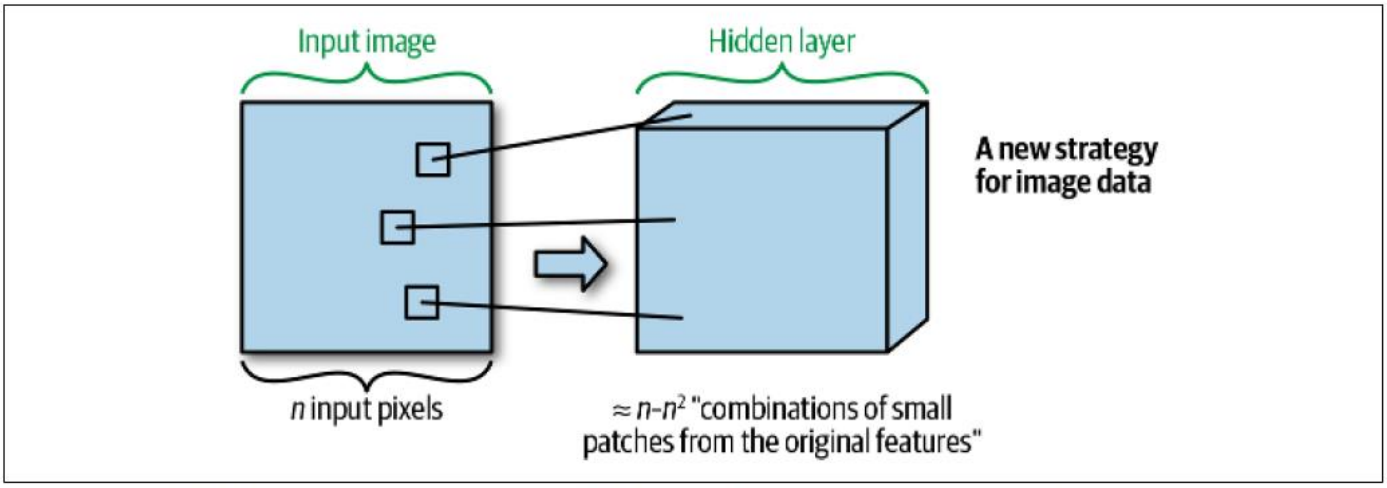


Figure 5-2. With image data, we can define each learned feature to be a function of a small patch of data, and thus define somewhere between n and n^2 output neurons

Having our neural network learn combinations of all of the input features—that is, combinations of all of the pixels in the input image—turns out to be very inefficient, since it ignores the insight described in the prior section: that most of the interesting combinations of features in images occur in these small patches.

What operation can we use to compute many combinations of the pixels from local patches of the input image?

The answer is the convolution operation.

1.2. The Convolution Operation

The convolution operation is a fundamental operation in deep learning, especially in convolutional neural networks (CNNs). CNNs are a type of neural network that is specifically designed for image processing and computer vision tasks.

CNNs use convolution operations to extract features from images. Features are patterns in the image that can be used to identify and classify objects. For example, some features of a face might include the eyes, nose, and mouth.

Convolution operations are performed by sliding a small filter over the image and computing the dot product of the filter and the image pixels at each location. The filter is typically a small square or rectangular array of weights. The result of the convolution operation is a new image that is smaller than the original image.

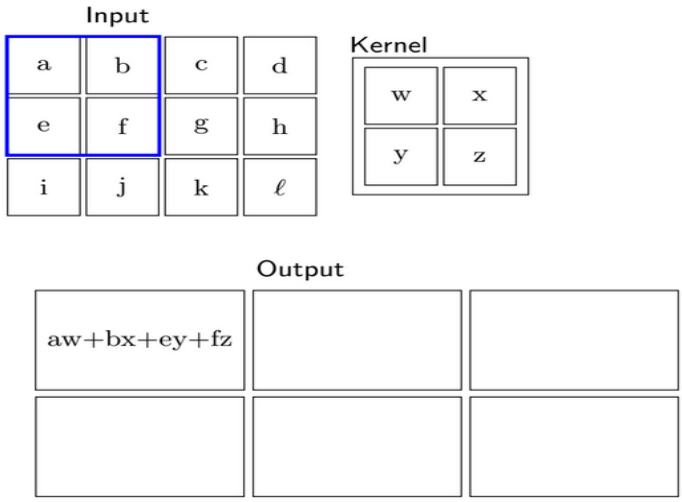
The new image contains the features that were extracted by the filter. For example, a filter might be designed to extract edge features from an image. The output of the convolution operation with this filter would be an image that highlights the edges in the original image.

CNNs typically have multiple convolutional layers, each of which uses a different filter to extract different features from the image. The output of the convolutional layers is then fed into a fully connected neural network, which performs classification or other tasks.

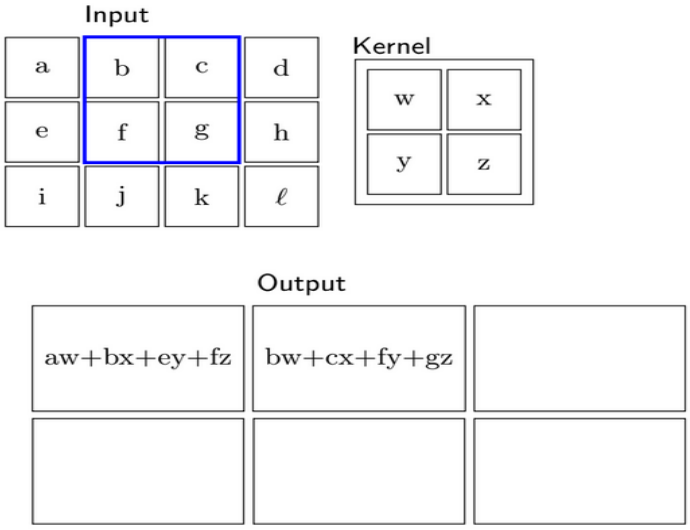
We compute the output(re-estimated value of current pixel) using the following formula:

$$S_{ij} = (I * K)_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I_{i+a,j+b} K_{a,b}$$

Here m refers to the number of rows(which is 2 in this case) and n refers to the number of columns(which is 2 i this case).

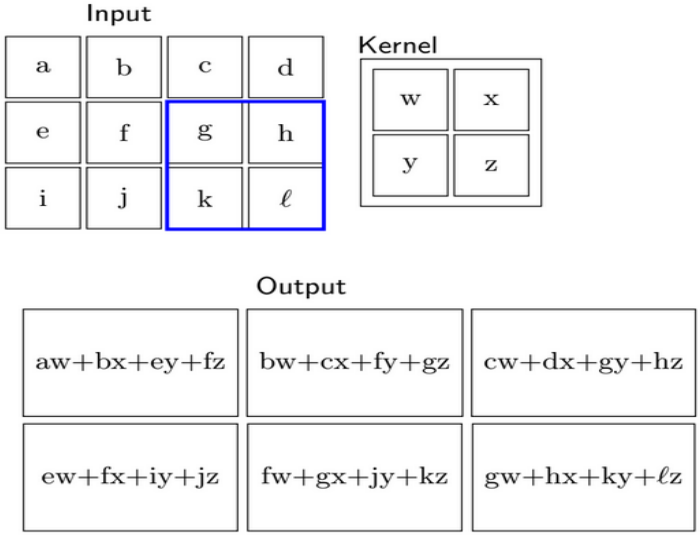


- Let us apply this idea to a toy example and see the results



- Let us apply this idea to a toy example and see the results

Similarly, we do the rest



- Let us apply this idea to a toy example and see the results

2. The Multichannel Convolution Operation

To review: convolutional neural networks differ from regular neural networks in that they create an order of magnitude more features, and in that each feature is a function of just a small patch from the input image.

Now we can get more specific: starting with *n* input pixels, the convolution operation just described will create *n* output features, one for each location in the input image.

What actually happens in a convolutional Layer in a neural network goes one step further: there, well create *f sets* of *n* features, *each* with a corresponding (initially random) set of weights defining a visual pattern whose detection at each location in the input image will be captured in the feature map.

These *f* feature maps will be created via *f* convolution operations. This is captured in Figure 5-3.

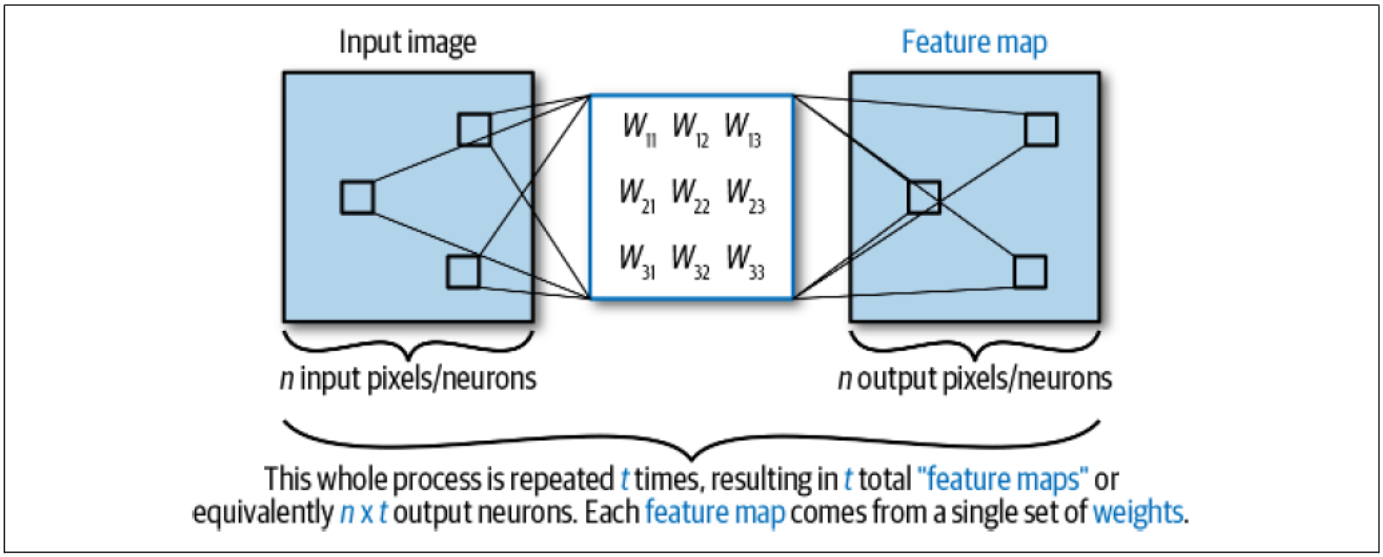


Figure 5-3. More specifically than before, for an input image with n pixels, we define an output with f feature maps, each of which has about the same size as the original image, for a total of n × f total output neurons for the image, each of which is a function of only a small patch of the original image

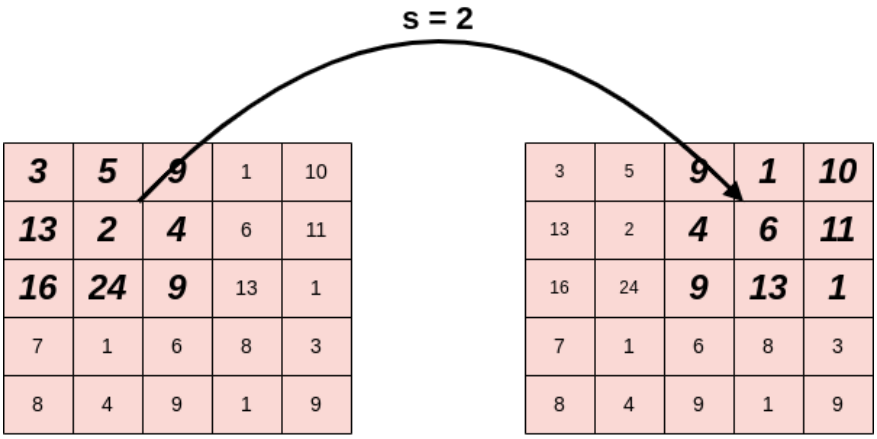
While each "set of features" detected by a particular set of weights is called a **feature map**, in the context of a convolutional Layer, the number of feature maps is referred to as the number of **channels** of the Layer—this is why the operation involved with the Layer is called the multichannel convolution. In addition, the f sets of weights Wi are called the convolutional filters.

Q) Relation between input size, output size and filter size

Stride

During convolution, the filter slides from left to right and from top to bottom until it passes through the entire input image. We define **stride as the step of the filter.** So, when we want to down sample the input image and end up with a smaller output, we set S>0.

Below, we can see example when:

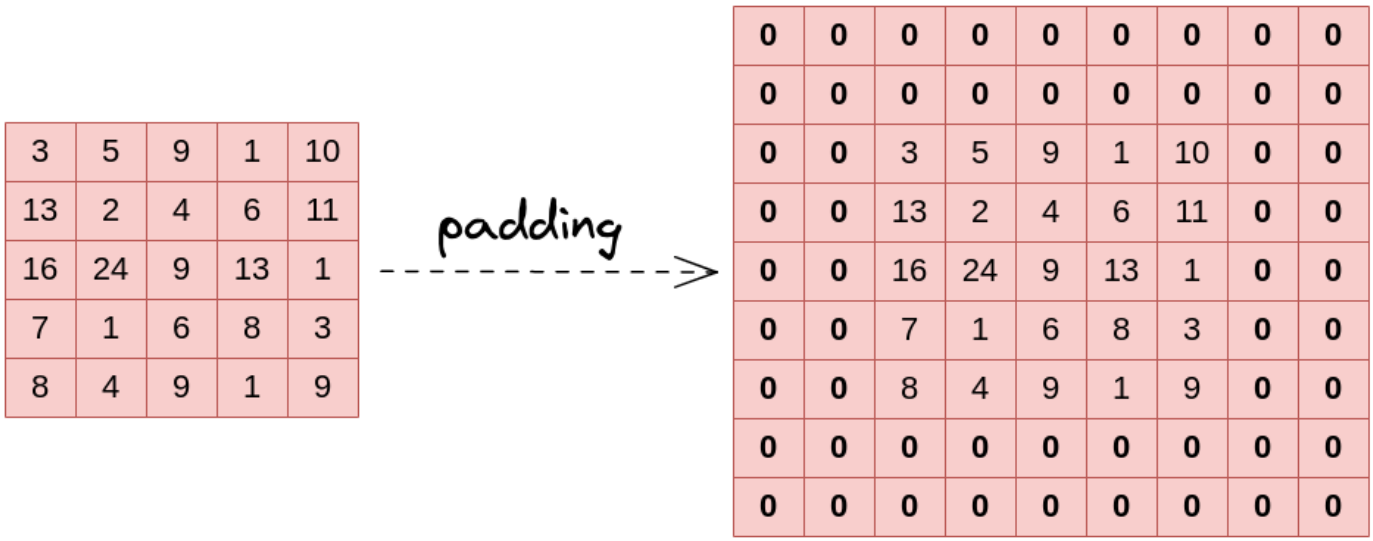


Padding

In a convolutional layer, we observe that **the pixels located on the corners and the edges are used much less than those in the middle.**

A simple and powerful solution to this problem is padding, which adds rows and columns of zeros to the input image. If we apply padding in an input image of size HXH ,the output image has dimensions (W+2P)X(H+2P).

Below we can see an example image before and after padding with $P = 2$. As we can see, the dimensions increased from 5×5 to 9×9:



By using padding in a convolutional layer, we increase the contribution of pixels at the corners and the edges to the learning procedure.

More Edge Detection

The type of filter that we choose helps to detect the vertical or horizontal edges. We can use the following filters to detect different edges:

1	0	-1
1	0	-1
1	0	-1

Vertical

1	1	1
0	0	0
-1	-1	-1

Horizontal

Some of the commonly used filters are:

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

The Sobel filter puts a little bit more weight on the central pixels. Instead of using these filters, we can create our own as well and treat them as a parameter which the model will learn using backpropagation.

To present the formula for computing the output size of a convolutional layer. We have the following input:

- An image of dimensions $W_{in} \times H_{in}$.
- A filter of dimensions $K \times K$.
- Stride S and padding P.

The output activation map will have the following dimensions:

- $\mathbf{W_{out} = \frac{W_{in}-K+2P}{S} + 1}$
- $\mathbf{H_{out} = \frac{H_{in}-K+2P}{S} + 1}$

If the output dimensions are not integers, it means that we haven't set the stride S correctly.

We have two exceptional cases:

- When there is no padding at all, the output dimensions are $\left(\frac{W_{in}-K}{S} + 1, \frac{H_{in}-K}{S} + 1\right)$.
- In case we want to keep the size of the input unchanged after the convolution layer, we apply same padding where $W_{out} = W_{in}$ and $H_{out} = H_{in}$. If s=1, we set $p = \frac{K-1}{2}$.

Example:

Let's suppose that we have an input image of size 125x49, a filter of size 5x5, padding P=2 and stride S=2. Then the output dimensions are the following:

- $W_{out} = \frac{125-5+2*2}{2} = \frac{124}{2} = 62$
- $H_{out} = \frac{49-5+2*2}{2} = \frac{48}{2} = 24$

So,the output activation map will have dimensions $(62, 24)$.

In fact these involve different aspects of parameters in a CNN.

1. Parameter sharing, means one parameter may be shared by more than one input/connection. So this reduces total amount of independent parameters. Parameters shared are non-zero.

2. Sparsity of connections means that some parameters are simply missing (ie are zero), **nothing to do with sharing same non-zero parameter**. Parameters in this case are zero, ignored. That means that **not necessarily all (potential) inputs to a layer are actually connected to that layer**, only some of them, rest are ignored. Thus sparsity of connections.

Q) Weight sharing is an old-school technique for reducing the number of weights in a network that must be trained. It is exactly what it sounds like: the reuse of weights on nodes that are close to one another in some way.

Weight sharing in CNNs

A typical application of weight sharing is in convolutional neural networks. CNNs work by passing a filter over the image input. For the trivial example of a 4x4 image and a 2x2 filter with a stride size of 2, this would mean that the filter (which has four weights, one per pixel) is applied four times, making for 16 weights total. A typical application of weight sharing is to share the same weights across all four filters.

In this context weight sharing has the following effects:

- It reduces the number of weights that must be learned (from 16 to 4, in this case), which reduces model training time and cost.
- It makes feature search insensitive to feature location in the image.

So we reduce training cost at the cost of model flexibility. Weight sharing is for all intents and purposes a form of regularization. And as with other forms of regularization, it can actually *increase* the performance of the model, in certain datasets with high feature location variance, by decreasing variance more than they increase bias.

Q) Pooling Layer

The pooling operation involves sliding a two-dimensional filter over each channel of feature map and summarising the features lying within the region covered by the filter.
For a feature map having dimensions n_h x n_w x n_c, the dimensions of output obtained after a pooling layer is

$$(n_h - f + 1) / s \times (n_w - f + 1)/s \times n_c$$

where,

-> n_h - height of feature map

-> n_w - width of feature map

-> n_c - number of channels in the feature map

-> **f** - size of filter

-> **s** - stride length

A common CNN model architecture is to have a number of convolution and pooling layers stacked one after the other.

Why to use Pooling Layers?

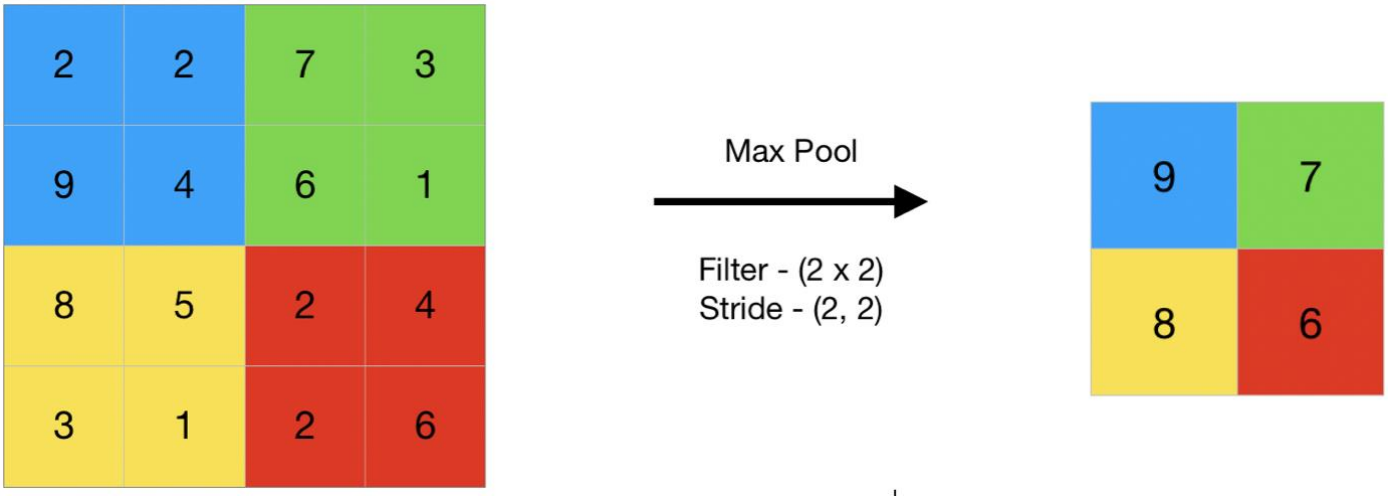
- Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.

The pooling layer summarises the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarised features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image.

Types of Pooling Layers

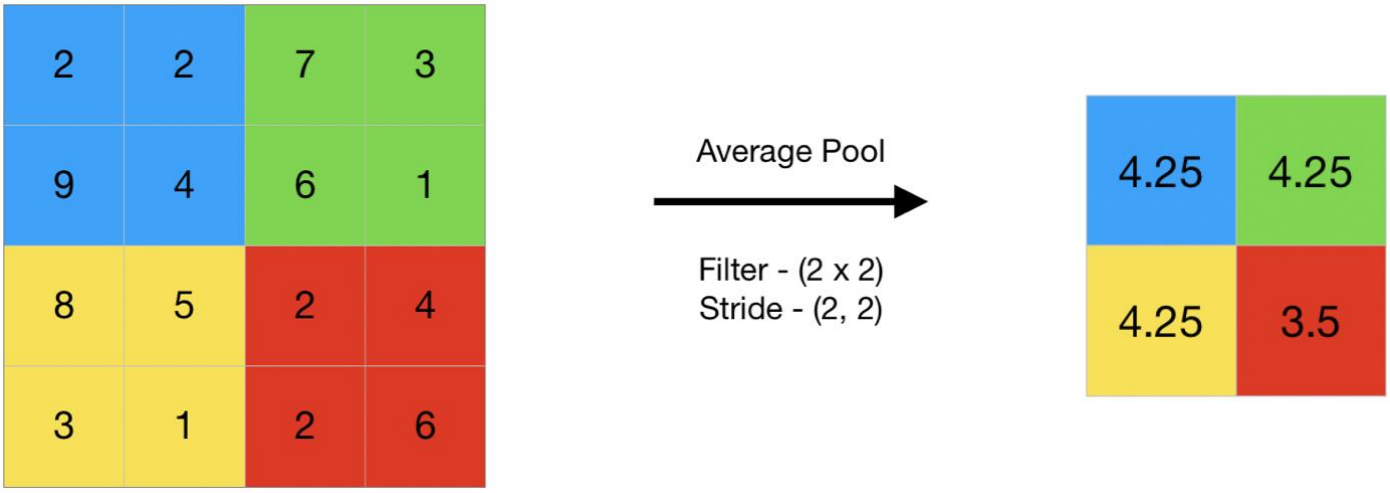
Max Pooling

Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.



Average Pooling

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.



In convolutional neural networks (CNNs), the pooling layer is a common type of layer that is typically added after convolutional layers. The pooling layer is used to reduce the spatial dimensions (i.e., the width and height) of the feature maps, while preserving the depth (i.e., the number of channels).

1. The pooling layer works by dividing the input feature map into a set of non-overlapping regions, called pooling regions. Each pooling region is then transformed into a single output value, which represents the presence of a particular feature in that region. The most common types of pooling operations are max pooling and average pooling.
2. In max pooling, the output value for each pooling region is simply the maximum value of the input values within that region. This has the effect of preserving the most salient features in each pooling region, while discarding less relevant information. Max pooling is often used in CNNs for object recognition tasks, as it helps to identify the most distinctive features of an object, such as its edges and corners.

3. In average pooling, the output value for each pooling region is the average of the input values within that region. This has the effect of preserving more information than max pooling, but may also dilute the most salient features. Average pooling is often used in CNNs for tasks such as image segmentation and object detection, where a more fine-grained representation of the input is required.

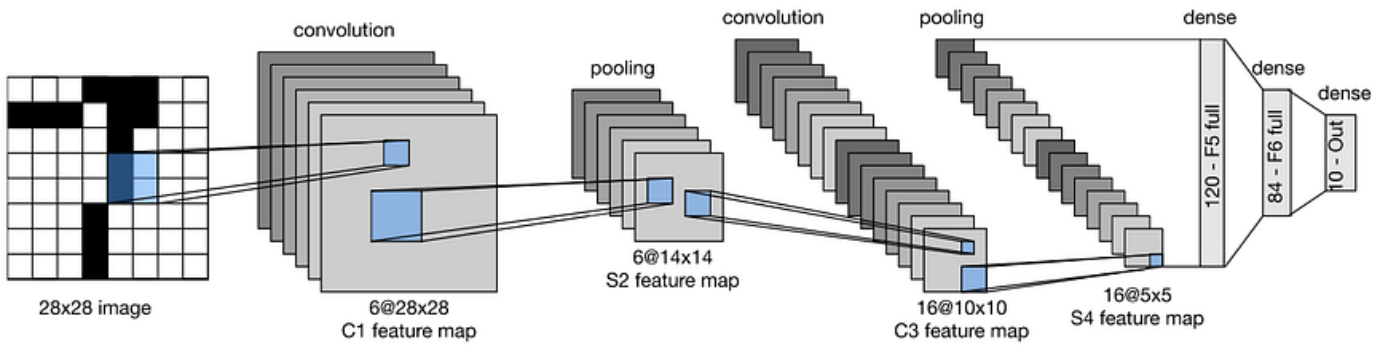
Pooling layers are typically used in conjunction with convolutional layers in a CNN, with each pooling layer reducing the spatial dimensions of the feature maps, while the convolutional layers extract increasingly complex features from the input. The resulting feature maps are then passed to a fully connected layer, which performs the final classification or regression task.

Q) LeNet-5 for handwritten character recognition

LeNet-5 is a convolutional neural network (CNN) architecture that was first proposed in 1998 for handwritten digit recognition. It is one of the earliest and most successful CNN architectures, and it has been used as a benchmark for many other CNN models.

LeNet-5 has a relatively simple architecture, consisting of the following layers:

- **Input layer:** This layer takes the input image, which is typically a 28x28 grayscale image of a handwritten digit.
- **Convolutional layer 1:** This layer extracts features from the input image using a set of convolution filters.
- **Pooling layer 1:** This layer reduces the dimensionality of the feature maps produced by the convolutional layer by downsampling them.
- **Convolutional layer 2:** This layer extracts more complex features from the feature maps produced by the first convolutional layer.
- **Pooling layer 2:** This layer further reduces the dimensionality of the feature maps produced by the second convolutional layer.
- **Fully connected layer:** This layer takes the flattened feature maps produced by the second pooling layer and produces a vector of outputs, one for each digit class.
- **Output layer:** This layer is a softmax layer that produces a probability distribution over the digit classes.



LeNet-5 can be trained to recognize handwritten digits by feeding it a dataset of labeled handwritten digit images. The network learns to extract features from the images that are discriminative for the different digit classes. Once the network is trained, it can be used to predict the digit class of a new handwritten digit image.

LeNet-5 has been shown to achieve high accuracy on the MNIST handwritten digit dataset, with an accuracy of over 99%. It has also been used to successfully recognize handwritten characters in other datasets, such as the USPS zip code dataset and the IAM handwritten text database.

While LeNet-5 is a relatively simple CNN architecture, it is still a powerful tool for handwritten character recognition. It is also a good architecture for beginners to learn about CNNs, as it is relatively easy to implement and train.

Q) How do we train a convolutional neural network ?

To train a convolutional neural network (CNN), you need to follow these steps:

1. **Collect a dataset of labeled images.** The dataset should be as large and diverse as possible, in order to train the network to generalize well to new images.
2. **Preprocess the images.** This may involve resizing the images, normalizing the pixel values, or converting the images to grayscale.
3. **Design the CNN architecture.** This involves choosing the number and type of layers, as well as the hyperparameters for each layer.
4. **Initialize the weights of the CNN.** This is typically done by randomly initializing the weights.
5. **Choose a loss function and optimizer.** The loss function measures how well the network is performing on the training data, and the optimizer updates the weights of the network to minimize the loss function.
6. **Train the CNN.** This involves feeding the training data to the network and updating the weights of the network using the optimizer.
7. **Evaluate the CNN.** Once the CNN is trained, you should evaluate its performance on a held-out test dataset. This will give you an idea of how well the network will generalize to new images.

Here is a more detailed explanation of each step:

Collect a dataset of labeled images: The dataset should be as large and diverse as possible, in order to train the network to generalize well to new images. The images should be labeled with the correct class, such as "cat", "dog", or "car". You can find pre-labeled datasets online, or you can collect your own dataset.

Preprocess the images: This may involve resizing the images to a consistent size, normalizing the pixel values, or converting the images to grayscale. Preprocessing the images helps to improve the performance of the CNN and makes it easier to train.

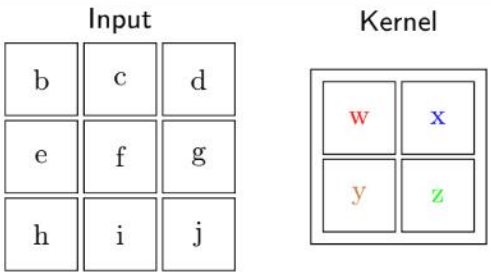
Design the CNN architecture: This involves choosing the number and type of layers, as well as the hyperparameters for each layer. The most common type of layer in a CNN is the convolutional layer. Convolutional layers extract features from the input images. Other common types of layers include pooling layers, fully connected layers, and output layers.

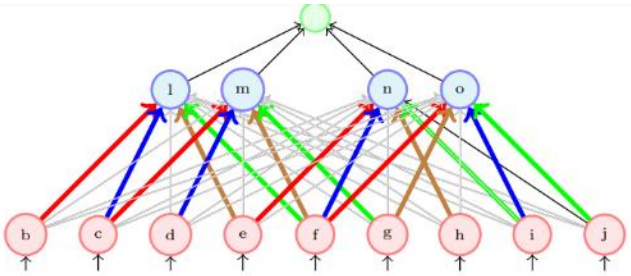
Initialize the weights of the CNN: This is typically done by randomly initializing the weights. However, there are also a number of techniques for initializing the weights in a way that improves the performance of the CNN.

Choose a loss function and optimizer: The loss function measures how well the network is performing on the training data. Common loss functions for CNNs include cross-entropy loss and mean squared error loss. The optimizer updates the weights of the network to minimize the loss function. Common optimizers for CNNs include Adam and stochastic gradient descent (SGD).

Train the CNN: This involves feeding the training data to the network and updating the weights of the network using the optimizer. The training process is typically repeated for a number of epochs, until the network converges to a good solution.

Evaluate the CNN: Once the CNN is trained, you should evaluate its performance on a held-out test dataset. This will give you an idea of how well the network will generalize to new images.





- A CNN can be implemented as a feedforward neural network
- wherein only a few weights(in color) are active
- the rest of the weights (in gray) are zero

- We can thus train a convolution neural network using backpropagation by thinking of it as a feedforward neural network with sparse connections

Q) AlexNet, ZF-Net, VGGNet, GoogLeNet and ResNet

AlexNet, ZF-Net, VGGNet, GoogLeNet, and ResNet are all convolutional neural networks (CNNs) that have achieved state-of-the-art results in image classification tasks.

AlexNet was the first CNN to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It has a relatively simple architecture, consisting of five convolutional layers followed by three fully connected layers. AlexNet was trained on a massive dataset of over 1.2 million images, and it achieved a top-5 error rate of 15.3% on the ILSVRC test set.

ZF-Net was inspired by AlexNet, but it introduced several improvements, such as the use of smaller convolutional filters and a deeper network architecture. ZF-Net achieved a top-5 error rate of 14.8% on the ILSVRC 2013 test set.

VGGNet is a family of CNNs that were developed by the University of Oxford. VGGNets are characterized by their use of very small convolutional filters and a very deep network architecture. VGGNet-16 achieved a top-5 error rate of 13.6% on the ILSVRC 2014 test set.

GoogLeNet is a CNN that was developed by Google. It is characterized by its use of the Inception module, which allows the network to learn multiple levels of abstraction in parallel. GoogLeNet achieved a top-5 error rate of 6.7% on the ILSVRC 2014 test set, which was a significant improvement over the previous state-of-the-art.

ResNet is a CNN that was developed by Microsoft. It is characterized by its use of residual blocks, which allow the network to learn deeper representations of the data without overfitting. ResNets have achieved state-of-the-art results on a wide range of image classification tasks, including the ILSVRC and COCO benchmarks.

All of these CNNs have played a significant role in the development of deep learning and computer vision. They have demonstrated the power of CNNs to learn complex representations of data and to solve challenging problems such as image classification.

Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	LeNet(8)	Yann LeCun et al			60 thousand
2012	AlexNet(7)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

Feature	AlexNet	ZF-Net	VGGNet	GoogLeNet	ResNet
Year	2012	2013	2014	2014	2015
Top-5 error on ILSVRC test set	15.3%	14.8%	13.6%	6.7%	3.57%
Depth	8	8	16	22	50
Number of parameters	61 million	58 million	138 million	6 million	25.6 million
Strengths	Simple architecture, effective for image classification	Deeper architecture, smaller convolutional filters	Very deep architecture, very small convolutional filters	Efficient architecture, learns multiple levels of abstraction in parallel	State-of-the-art results on a wide range of image classification tasks
Weaknesses	Relatively large number of parameters, difficult to train	Deeper architecture, more difficult to train	Very deep architecture, difficult to train	Requires more computational resources to train	More difficult to train than other CNNs

Q) What are Filters/Kernels?

- A filter provides a measure for how close a patch or a region of the input resembles a feature. A feature may be any prominent aspect – a vertical edge, a horizontal edge, an arch, a diagonal, etc.
- A filter acts as a single template or pattern, which, when convolved across the input, finds similarities between the stored template & different locations/regions in the input image.
- Let us consider an example of detecting a vertical edge in the input image.
- Each column of the 4×4 output matrix looks at exactly three columns & three rows (the coloured boxes show the output of the filter as it moves over the input image). The values in the output matrix represent the change in the intensity along the horizontal direction w.r.t the columns in the input image.
- The output image has the value 0 in the 1st & last column. It means there is no change in intensity in the first three columns & the previous three columns of the input image. On the other hand, the output

is 30 in the 2nd & 3rd column, indicating a change in the intensity of the corresponding columns of the input image.

Dimensions of the Convolved Output?

If the input image size is 'n x n' & filter size is 'f x f ', then after convolution, the size of the output image is: (Size of input image – filter size + 1)x (Size of input image – filter size + 1).

How is the Filter Size Decided?

By convention, the value of 'f,' i.e. filter size, is usually odd in computer vision. This might be because of 2 reasons:

- If the value of 'f' is even, we may need asymmetric padding (Please refer above eqn. 1). Let us say that the size of the filter i.e. 'f' is 6. Then by using equation 1, we get a padding size of 2.5, which does not make sense.
- The 2nd reason for choosing an odd size filter such as a 3×3 or a 5×5 filter is we get a central position & at times it is nice to have a distinguisher.

Multiple Filters for Multiple Features

- We can use multiple filters to detect various features simultaneously. Let us consider the following example in which we see vertical edge & curve in the input RGB image. We will have to use two different filters for this task, and the output image will thus have two feature maps.

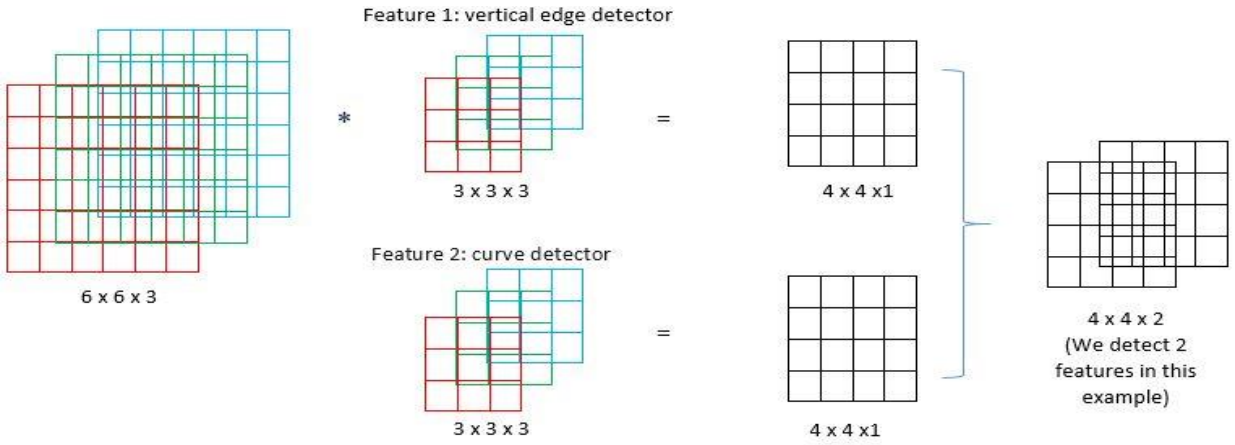


Fig: Convolution using multiple filters

- Let us understand the dimensions mathematically,

$$(n \times n \times n_c) * (f \times f \times n_c) * (f \times f \times n_c) \longrightarrow (n-f+1) \times (n-f+1) \times \text{no. of filters}$$

Here are some examples of filters that can be used in CNNs:

- **Edge detectors:** These filters are designed to detect edges in images. They can be used to extract features such as horizontal edges, vertical edges, and diagonal edges.
- **Corner detectors:** These filters are designed to detect corners in images. They can be used to extract features such as right angles, acute angles, and obtuse angles.
- **Texture detectors:** These filters are designed to detect textures in images. They can be used to extract features such as bumps, grooves, and patterns.

Introduction to RNN, RNN Code

……

1. Introduction to RNN

1.1. Sequence Learning Problems

Sequence learning problems are different from other machine learning problems in two key ways:

- The inputs to the model are not of a fixed size.
- The inputs to the model are dependent on each other.

Examples of sequence learning problems include:

- Auto completion
- Part-of-speech tagging
- Sentiment analysis
- Video classification

Recurrent neural networks (RNNs) are a type of neural network that are well-suited for solving sequence learning problems. RNNs work by maintaining a hidden state that is updated at each time step. The hidden state captures the information from the previous inputs, which allows the model to predict the next output.

Example:

Consider the task of auto completion. Given a sequence of characters, we want to predict the next character. For example, given the sequence "d", we want to predict the next character, which is "e".

An RNN would solve this problem by maintaining a hidden state. The hidden state would be initialized with the information from the first input character, "d". Then, at the next time step, the RNN would take the current input character, "e", and the hidden state as input and produce a prediction for the next character. The hidden state would then be updated with the new information.

This process would be repeated until the end of the sequence. At the end of the sequence, the RNN would output the final prediction.

Advantages of RNNs for sequence learning problems:

- RNNs can handle inputs of any length.
- RNNs can learn long-term dependencies between the inputs in a sequence.

Disadvantages of RNNs:

- RNNs can be difficult to train.
- RNNs can be susceptible to vanishing and exploding gradients.

RNNs are a powerful tool for solving sequence learning problems. They have been used to achieve state-of-the-art results in many tasks, such as machine translation, text summarization, and speech recognition.

1.2. Recurrent Neural Networks

Recurrent neural networks (RNNs) are a type of neural network that are well-suited for solving sequence learning problems. RNNs work by maintaining a hidden state that is updated at each time step. The hidden state captures the information from the previous inputs, which allows the model to predict the next output.

RNNs have several advantages over other types of neural networks for sequence learning problems:

- RNNs can handle inputs of any length.
- RNNs can learn long-term dependencies between the inputs in a sequence.
- RNNs can be used to solve a wide variety of sequence learning problems, such as natural language processing, machine translation, and speech recognition.

How to model sequence learning problems with RNNs:

To model a sequence learning problem with an RNN, we first need to define the function that the RNN will compute at each time step. The function should take as input the current input and the hidden state from the previous time step, and output the next hidden state and the prediction for the current time step.

Once we have defined the function, we can train the RNN using backpropagation through time (BPTT). BPTT is a specialized training algorithm for RNNs that allows us to train the network even though it has recurrent connections.

Examples of sequence learning problems that can be solved with RNNs:

- Natural language processing: tasks such as part-of-speech tagging, named entity recognition, and machine translation.
- Speech recognition: tasks such as transcribing audio to text and generating text from speech.
- Video processing: tasks such as video classification and captioning.
- Time series analysis: tasks such as forecasting and anomaly detection.

How RNNs solve the problems on the wishlist:

- **The same function is executed at every time step:** This is achieved by sharing the same network parameters at every time step.
- **The model can handle inputs of arbitrary length:** This is because the RNN can keep updating its hidden state based on the previous inputs, regardless of the length of the input sequence.
- **The model can learn long-term dependencies between the inputs in a sequence:** This is because the RNN's hidden state can capture information from the previous inputs, even if they are many time steps ago.

1.3. Backpropagation through time (BPTT)

BPTT is a training algorithm for recurrent neural networks (RNNs). It is used to compute the gradients of the loss function with respect to the RNN's parameters, which are then used to update the parameters using gradient descent.

To compute the gradients using BPTT, we need to first compute the explicit derivative of the loss function with respect to the RNN's parameters. This is done by treating all of the other inputs to the RNN as constants.

However, RNNs also have implicit dependencies, which means that the output of the RNN at a given time step depends on the outputs of the RNN at previous time steps. This makes it difficult to compute the gradients using the explicit derivative alone.

To address this problem, BPTT uses the chain rule to recursively compute the implicit derivatives of the loss function with respect to the RNN's parameters. This involves summing over all of the paths from the loss function to each parameter, where each path is a sequence of RNN outputs and weights.

BPTT can be computationally expensive, but it is a powerful tool for training RNNs. It has been used to achieve state-of-the-art results on a variety of sequence learning tasks, such as natural language processing, machine translation, and speech recognition.

Example:

Consider the following RNN, which is used to predict the next character in a sequence:

s_t = W * s_{t-1} + x_t

y_t = softmax(V * s_t)

where:

- s_t is the hidden state of the RNN at time step t
- x_t is the input at time step t
- y_t is the output at time step t
- W and V are the RNN's parameters

Suppose we want to compute the gradient of the loss function with respect to the weight W. Using BPTT, we can do this as follows:

```
# Compute the explicit derivative

d_loss_dw = s_{t-1}

# Compute the implicit derivative

for i in range(t - 2, -1, -1):
    d_loss_dw += s_i * W^T * d_loss_dw
```

The implicit derivative is computed by recursively summing over all of the paths from the loss function to the weight W. Each path is a sequence of RNN outputs and weights, and the derivative for each path is computed using the chain rule.

Once we have computed the explicit and implicit derivatives, we can simply sum them together to get the total derivative of the loss function with respect to the weight W. This derivative can then be used to update the weight W using gradient descent.

Challenges:

BPTT can be computationally expensive, especially for RNNs with many layers or long sequences. However, there are a number of techniques that can be used to improve the efficiency of BPTT, such as truncated BPTT and gradient clipping.

Another challenge with BPTT is that it can be sensitive to the initialization of the RNN's parameters. If the parameters are not initialized carefully, the RNN may not learn to perform the desired task.

1.4. The problem of Exploding and Vanishing Gradients

The problem of vanishing and exploding gradients is a common problem when training recurrent neural networks (RNNs). It occurs because the gradients of the loss function with respect to the RNN's parameters can become very small or very large as the backpropagation algorithm progresses. This can make it difficult for the RNN to learn to perform the desired task.

There are two main reasons why vanishing and exploding gradients can occur:

1. **Bounded activations:** RNNs typically use bounded activation functions, such as the sigmoid or tanh function. This means that the derivatives of the activation functions are also bounded. This can lead to vanishing gradients, especially if the RNN has a large number of layers.
2. **Product of weights:** The gradients of the loss function with respect to the RNN's parameters are computed by multiplying together the gradients of the activations at each layer. This means that if the gradients of the activations are small or large, the gradients of the parameters will also be small or large.

Vanishing and exploding gradients can be a major problem for training RNNs. If the gradients vanish, the RNN will not be able to learn to perform the desired task. If the gradients explode, the RNN will learn very quickly, but it will likely overfit the training data and not generalize well to new data.

There are a number of techniques that can be used to address the problem of vanishing and exploding gradients, such as:

- **Truncated backpropagation:** Truncated backpropagation only backpropagates the gradients through a fixed number of layers. This helps to prevent the gradients from vanishing.
- **Gradient clipping:** Gradient clipping normalizes the gradients so that their magnitude does not exceed a certain threshold. This helps to prevent the gradients from exploding.
- **Weight initialization:** The way that the RNN's parameters are initialized can have a big impact on the problem of vanishing and exploding gradients. It is important to initialize the parameters in a way that prevents the gradients from becoming too small or too large.

Truncated backpropagation is a common technique used to address the problem of vanishing and exploding gradients in recurrent neural networks (RNNs). However, it is not the only solution.

Another common solution is to use **gated recurrent units (GRUs)** or **long short-term memory (LSTM) cells**. These units are specifically designed to deal with the problem of vanishing and exploding gradients.

GRUs and LSTMs work by using gates to control the flow of information through the RNN. This allows the RNN to learn long-term dependencies in the data without the problem of vanishing gradients.

GRUs and LSTMs have been shown to be very effective for training RNNs on a variety of tasks, such as natural language processing, machine translation, and speech recognition.

1.5. Long Short Term Memory(LSTM) and Gated Recurrent Units(GRUs)

Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) are two types of recurrent neural networks (RNNs) that are specifically designed to learn long-term dependencies in sequential data. They are both widely used in a variety of tasks, including natural language processing, machine translation, speech recognition, and time series forecasting.

Both LSTMs and GRUs use a gating mechanism to control the flow of information through the network. This allows them to learn which parts of the input sequence are important to remember and which parts can be forgotten.

LSTM Architecture

An LSTM cell has three gates: an input gate, a forget gate, and an output gate.

- The input gate controls how much of the current input is added to the cell state.
- The forget gate controls how much of the previous cell state is forgotten.
- The output gate controls how much of the cell state is output to the next cell in the sequence.

The LSTM cell also has a cell state, which is a long-term memory that stores information about the previous inputs. The cell state is updated at each time step based on the input gate, forget gate, and output gate.

GRU Architecture

A GRU cell has two gates: a reset gate and an update gate.

- The reset gate controls how much of the previous cell state is forgotten.
- The update gate controls how much of the previous cell state is combined with the current input to form the new cell state.

The GRU cell does not have a separate output gate. Instead, the output of the GRU cell is simply the updated cell state.

Comparison of LSTMs and GRUs

LSTMs and GRUs are very similar in terms of their performance on most tasks. However, there are a few key differences between the two architectures:

- LSTMs have more gates and parameters than GRUs, which makes them more complex and computationally expensive to train.
- GRUs are generally faster to train and deploy than LSTMs.
- GRUs are more robust to noise in the input data than LSTMs.

Which one to choose?

The best choice of architecture for a particular task depends on a number of factors, including the size and complexity of the dataset, the available computing resources, and the specific requirements of the task.

In general, LSTMs are recommended for tasks where the input sequences are very long or complex, or where the task requires a high degree of accuracy. GRUs are a good choice for tasks where the input sequences are shorter or less complex, or where speed and efficiency are important considerations.

2. RNN Code

```
import keras

# Define the model

model = keras.Sequential([

    keras.layers.LSTM(128, input_shape=(10, 256)),

    keras.layers.Dense(64, activation='relu'),

    keras.layers.Dense(1, activation='sigmoid')

])

# Compile the model

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

# Train the model

model.fit(x_train, y_train, epochs=10)

# Evaluate the model

model.evaluate(x_test, y_test)

# Make predictions

predictions = model.predict(x_test)
```

This code defines a simple RNN model with one LSTM layer, one dense layer, and one output layer. The LSTM layer has 128 hidden units, and the dense layer has 64 hidden units. The output layer has a single unit, and it uses the sigmoid activation function to produce a probability score.

The model is compiled using the binary cross-entropy loss function and the Adam optimizer. The model is then trained on the training data for 10 epochs.

Once the model is trained, it can be evaluated on the test data to assess its performance. The model can also be used to make predictions on new data.

Here is an example of how to use the model to make predictions:

```
# Make predictions on a new data sample

x_new = [[[0.1, 0.2], [0.3, 0.4], [0.5, 0.6]]]

# Get the prediction

prediction = model.predict(x_new)

# Print the prediction

print(prediction)
```

This code will print the prediction for the new data sample, which is a probability score between 0 and 1. A probability score closer to 1 means that the model is more confident in the prediction.

This is just a simple example of RNN code, and there are many other ways to implement RNNs in Python. For more complex tasks, you may need to use a different RNN architecture or add additional layers to the model.

Chapter-III

PyTorch Tensors: Deep Learning with PyTorch, CNN in PyTorch

……..

PyTorch is **an optimized Deep Learning tensor library based on Python and Torch and is mainly used for applications using GPUs and CPUs**. PyTorch is favored over other Deep Learning frameworks like TensorFlow and Keras since it uses dynamic computation graphs and is completely Pythonic.

Why is PyTorch used for deep learning?

It is open source, and is based on the popular Torch library. PyTorch is designed to provide good flexibility and high speeds for deep neural network implementation. PyTorch is different from other deep learning frameworks in that it uses dynamic computation graphs.

3.1.Features

The major features of PyTorch are mentioned below −

Easy Interface − PyTorch offers easy to use API; hence it is considered to be very simple to operate and runs on Python. The code execution in this framework is quite easy.

Python usage − This library is considered to be Pythonic which smoothly integrates with the Python data science stack. Thus, it can leverage all the services and functionalities offered by the Python environment.

Computational graphs − PyTorch provides an excellent platform which offers dynamic computational graphs. Thus a user can change them during runtime. This is highly useful when a developer has no idea of how much memory is required for creating a neural network model.

PyTorch is known for having three levels of abstraction as given below −

- Tensor − Imperative n-dimensional array which runs on GPU.
- Variable − Node in computational graph. This stores data and gradient.
- Module − Neural network layer which will store state or learnable weights.

Advantages of PyTorch

The following are the advantages of PyTorch −

- It is easy to debug and understand the code.
- It includes many layers as Torch.
- It includes lot of loss functions.
- It can be considered as NumPy extension to GPUs.
- It allows building networks whose structure is dependent on computation itself.

3.2.TensorFlow vs. PyTorch

We shall look into the major differences between TensorFlow and PyTorch below –

PyTorch	TensorFlow
PyTorch is closely related to the lua-based Torch framework which is actively used in Facebook.	TensorFlow is developed by Google Brain and actively used at Google.
PyTorch is relatively new compared to other competitive technologies.	TensorFlow is not new and is considered as a to-go tool by many researchers and industry professionals.
PyTorch includes everything in imperative and dynamic manner.	TensorFlow includes static and dynamic graphs as a combination.
Computation graph in PyTorch is defined during runtime.	TensorFlow do not include any run time option.
PyTorch includes deployment featured for mobile and embedded frameworks.	TensorFlow works better for embedded frameworks.

3.3. Deep Learning with PyTorch

PyTorch elements are the building blocks of PyTorch models. These elements are:

- **Model:** A model is a representation of a machine learning algorithm. It is made up of layers and parameters.
- **Layer:** A layer is a unit of computation in a neural network. It performs a specific mathematical operation on the input data.
- **Optimizer:** An optimizer is an algorithm that updates the model's parameters during training.
- **Loss:** A loss function measures the error between the model's predictions and the ground truth labels.

Models are created using the torch.nn.Module class. Layers are created using the different classes provided by the torch.nn module. For example, to create a linear layer, you would use the torch.nn.Linear class.

Optimizers are created using the classes provided by the torch.optim module. For example, to create an Adam optimizer, you would use the torch.optim.Adam class.

Loss functions are created using the classes provided by the torch.nn.functional module. For example, to create a mean squared error loss function, you would use the torch.nn.functional.mse_loss function.

Once you have created the model, layers, optimizer, and loss function, you can train the model using the following steps:

1. Forward pass: The input data is passed through the model to produce predictions.
2. Loss calculation: The loss function is used to calculate the error between the predictions and the ground truth labels.

3. Backward pass: The gradients of the loss function with respect to the model's parameters are calculated.
4. Optimizer step: The optimizer uses the gradients to update the model's parameters.

This process is repeated for a number of epochs until the model converges and achieves the desired performance.

Here is a simple example of a PyTorch model:

```
import torch

class LinearModel(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearModel, self).__init__()

        self.linear = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        output = self.linear(x)

        return output

# Create the model
model = LinearModel(10, 1)

# Create the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Create the loss function
loss_fn = torch.nn.functional.mse_loss

# Train the model
...
```

This code defines a simple linear model with one input layer and one output layer. The model is trained using the Adam optimizer and the mean squared error loss function.

3.4. RNN with PyTorch

PyTorch is a Python library for machine learning. It is based on Torch, a scientific computing framework for Lua. PyTorch is popular for its flexibility and ease of use. It is also widely used in research and industry.

To implement RNNs in PyTorch, you can use the **torch.nn.RNN** module. This module provides a number of different RNN architectures, including LSTM and GRU.

Here is a simple example of how to implement an LSTM in PyTorch:

This code defines a simple LSTM model with one input layer, one LSTM layer, and one output layer. The LSTM layer has 128 hidden units.

The model is trained using the model.fit() method. The model can then be used to make predictions on new data using the model.predict() method.

For more complex tasks, you may need to use a different RNN architecture or add additional layers to the model. You can also use PyTorch to implement bidirectional RNNs, stacked RNNs, and other advanced RNN architectures.

PyTorch also provides a number of tools for training and evaluating RNNs, such as the torch.optim module and the torch.nn.functional module.

```
import torch

class LSTM(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(LSTM, self).__init__()

        self.lstm = torch.nn.LSTM(input_size, hidden_size, num_layers)

    def forward(self, x):
        output, (hn, cn) = self.lstm(x)

        return output

# Define the model
model = LSTM(10, 128, 1)

# Train the model
...

# Make predictions
...
```

3.4. CNN with PyTorch

Convolutional neural networks (CNNs) are a type of neural network that are specifically designed to work with image data. CNNs are able to learn spatial features in images, which makes them very effective for tasks such as image classification, object detection, and image segmentation.

PyTorch is a popular Python library for machine learning. It provides a number of features that make it easy to build, train, and deploy CNNs.

To implement a CNN in PyTorch, you can use the torch.nn.Conv2d layer. This layer performs a convolution operation on the input data. The convolution operation is a mathematical operation that extracts features from the input data.

CNNs also use pooling layers to reduce the spatial size of the input data. This helps to reduce the number of parameters in the network and makes it more efficient to train.

Here is a simple example of a CNN in PyTorch:

```
import torch

class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()


        # Define the convolutional layers
        self.conv1 = torch.nn.Conv2d(3, 6, 5)
        self.conv2 = torch.nn.Conv2d(6, 16, 5)


        # Define the pooling layers
        self.pool1 = torch.nn.MaxPool2d(2, 2)
        self.pool2 = torch.nn.MaxPool2d(2, 2)
```

```
        # Define the fully connected layers
        self.fc1 = torch.nn.Linear(16 * 5 * 5, 120)
        self.fc2 = torch.nn.Linear(120, 84)
        self.fc3 = torch.nn.Linear(84, 10)

    def forward(self, x):
        # Pass the input data through the convolutional layers
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)

        # Flatten the output of the convolutional layers
        x = x.view(-1, 16 * 5 * 5)

        # Pass the flattened output through the fully connected layers
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
# Create the model
model = CNN()
# Train the model
...
```

This code defines a simple CNN with two convolutional layers, two pooling layers, and three fully connected layers. The convolutional layers have 6 and 16 filters, respectively. The pooling layers have a kernel size of 2x2 and a stride of 2. The fully connected layers have 120, 84, and 10 units, respectively.

The model is trained using the model.fit() method. The model can then be used to make predictions on new data using the model.predict() method.

For more complex tasks, you may need to use a different CNN architecture or add additional layers to the model. You can also use PyTorch to implement other types of neural networks, such as recurrent neural networks (RNNs) and long short-term memory (LSTM) networks.

UNIT V

Interactive Applications of Deep Learning: Machine Vision, Natural Language processing, Generative Adversial Networks, Deep Reinforcement Learning.

Deep Learning Research: Autoencoders, Deep Generative Models: Boltzmann Machines Restricted Boltzmann Machines, Deep Belief Networks.

……

Interactive Applications of Deep Learning

Chapter-1:Machine Vision

1.1.Convolutional Neural Networks

a) The Two-Dimensional Structure of Visual Imagery

b) Computational Complexity

c) Convolutional Layers

c) Multiple Filters

d) A Convolutional Example

e) Convolutional Filter Hyperparameters

1.2.Pooling Layers

1.3.LeNet-5 in Keras

1.4.AlexNet and VGGNet in Keras

1.5.Residual Networks

a) Vanishing Gradients: The Bête Noire of Deep CNNs

b) Residual Connections

c) ResNet

1.6.Applications of Machine Vision

a) Object Detection

b) Image Segmentation

c) Transfer Learning

d) Capsule Networks

Chapter-II: Natural Language processing

2.1. Preprocessing Natural Language Data

a) Tokenization

b) Converting All Characters to Lowercase

c) Removing Stop Words and Punctuation

d) Stemming

e) Handling n-grams

f) Preprocessing the Full Corpus

2.2. Creating Word Embeddings with word2vec

a) The Essential Theory Behind word2vec

b) Evaluating Word Vectors

c) Running word2vec

d) Plotting Word Vectors

2.3. The Area under the ROC Curve

a) The Confusion Matrix

b) Calculating the ROC AUC Metric

2.4.Natural Language Classification with Familiar Networks

a) Loading the IMDb Film Reviews

b) Examining the IMDb Data

c) Standardizing the Length of the Reviews

d) Dense Network

e) Convolutional Networks

2.5.Networks Designed for Sequential Data

a) Recurrent Neural Networks

b) Long Short-Term Memory Units

c) Bidirectional LSTMs

d) Stacked Recurrent Models

e) Seq2seq and Attention

e) Transfer Learning in NLP

2.6.Non-sequential Architectures: The Keras Functional API

Chapter-III: Generative Adversial Networks

3.1.Essential GAN Theory

3.3.The Quick, Draw! Dataset

3.4.The Discriminator Network

3.5.The Generator Network

3.6.The Adversarial Network

3.7.GAN Training

Chapter-IV: Deep Reinforcement Learning

4.1.Essential Theory of Reinforcement Learning

a) The Cart-Pole Game

b) Markov Decision Processes

c) The Optimal Policy

4.2.Essential Theory of Deep Q-Learning Networks

a) Value Functions

b) Q-Value Functions

c) Estimating an Optimal Q-Value

4.3.Defining a DQN Agent

a) Initialization Parameters

b) Building the Agent's Neural Network Model

c) Remembering Gameplay

d) Training via Memory Replay

e) Selecting an Action to Take

f) Saving and Loading Model Parameters

4.4.Interacting with an OpenAI Gym Environment

4.5.Hyperparameter Optimization with SLM Lab

4.6.Agents Beyond DQN

a) Policy Gradients and the REINFORCE Algorithm

b) The Actor-Critic Algorithm

……..

Deep Learning Research

Chapter-I: Autoencoders

1.1 Undercomplete Autoencoders

1.2 Regularized Autoencoders

1.3 Representational Power, Layer Size and Depth

1.4 Stochastic Encoders and Decoders

1.5 Denoising Autoencoders

1.6 Learning Manifolds with Autoencoders

1.7 Contractive Autoencoders

1.8 Predictive Sparse Decomposition

1.9 Applications of Autoencoders

Chapter-II: Deep Generative Models

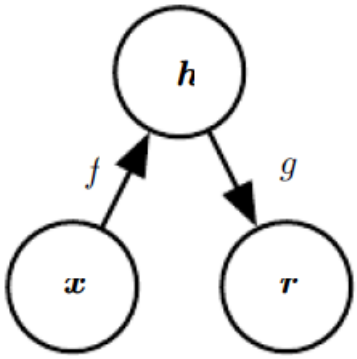
2.1.Boltzmann Machines

2.2.Restricted Boltzmann Machines

2.3.Deep Belief Networks

Chapter-I: Autoencoders

- An autoencoder is a neural network that is trained to attempt to copy its input to its output.
- Internally, it has a hidden layer h that describes a code used to represent the input.
- The network may be viewed as consisting of two parts: an encoder function h = f (x) and a decoder that produces a reconstruction r = g(h). This architecture is presented in below figure



An autoencoder is a type of artificial neural network that is trained to reconstruct its input. It does this by learning a hidden representation of the input, and then using this hidden representation to generate a reconstructed output.

Autoencoders are typically used for unsupervised learning tasks, such as dimensionality reduction, anomaly detection, and feature extraction. They can also be used for supervised learning tasks, such as classification and regression, by pre-training the autoencoder on an unlabeled dataset and then using the learned hidden representations as input to a supervised learning model.

Autoencoders consist of two main parts: an encoder and a decoder. The encoder takes the input data and compresses it into a hidden representation. The decoder then takes the hidden representation and reconstructs the output data.

Autoencoders are trained by minimizing the difference between the input data and the reconstructed output data. This is typically done using a loss function, such as the mean squared error.

Once an autoencoder is trained, it can be used to generate new data that is similar to the training data. This can be useful for tasks such as data augmentation and image generation.

Here is a simple example of how to use an autoencoder to reduce the dimensionality of a dataset:

1. Train an autoencoder on the dataset.
2. Use the encoder to generate a hidden representation of the dataset.
3. Discard the original dataset and only keep the hidden representations.
4. Use the hidden representations as input to a supervised learning model, such as a logistic regression classifier.

This approach can be used to reduce the dimensionality of the dataset while preserving the most important information. This can improve the performance of the supervised learning model, especially if the dataset is large or noisy.

1.1. Undercomplete Autoencoder

The objective of undercomplete autoencoder is to capture the most important features present in the data. Undercomplete autoencoders have a smaller dimension for hidden layer compared to the input layer. This helps to obtain important features from the data. It minimizes the loss function by penalizing the g(f(x)) for being different from the input x.

Advantages-

- Undercomplete autoencoders do not need any regularization as they maximize the probability of data rather than copying the input to the output.

Drawbacks-

- Using an overparameterized model due to lack of sufficient training data can create overfitting.

1.2. Regularized autoencoders

Regularized autoencoders are a type of autoencoder that incorporates regularization techniques to prevent overfitting and improve generalization. Overfitting occurs when a model learns the training data too well and is unable to generalize to new data. Regularization techniques help to prevent this by constraining the model to learn simpler representations of the data.

There are three main types of regularized autoencoders:

1.2.1) Denoising Autoencoder

Denoising autoencoders create a corrupted copy of the input by introducing some noise. This helps to avoid the autoencoders to copy the input to the output without learning features about the data.

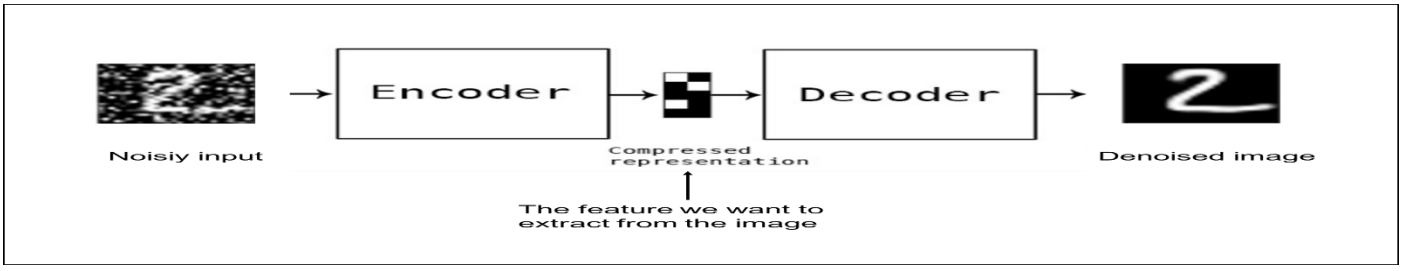
These autoencoders take a partially corrupted input while training to recover the original undistorted input. The model learns a vector field for mapping the input data towards a lower dimensional manifold which describes the natural data to cancel out the added noise.

Advantages-

- It was introduced to achieve good representation. Such a representation is one that can be obtained robustly from a corrupted input and that will be useful for recovering the corresponding clean input.
- Corruption of the input can be done randomly by making some of the input as zero. Remaining nodes copy the input to the noised input.
- Minimizes the loss function between the output node and the corrupted input.
- Setting up a single-thread denoising autoencoder is easy.

Drawbacks-

- To train an autoencoder to denoise data, it is necessary to perform preliminary stochastic mapping in order to corrupt the data and use as input.
- This model isn't able to develop a mapping which memorizes the training data because our input and target output are no longer the same.



1.2.2) Sparse Autoencoder

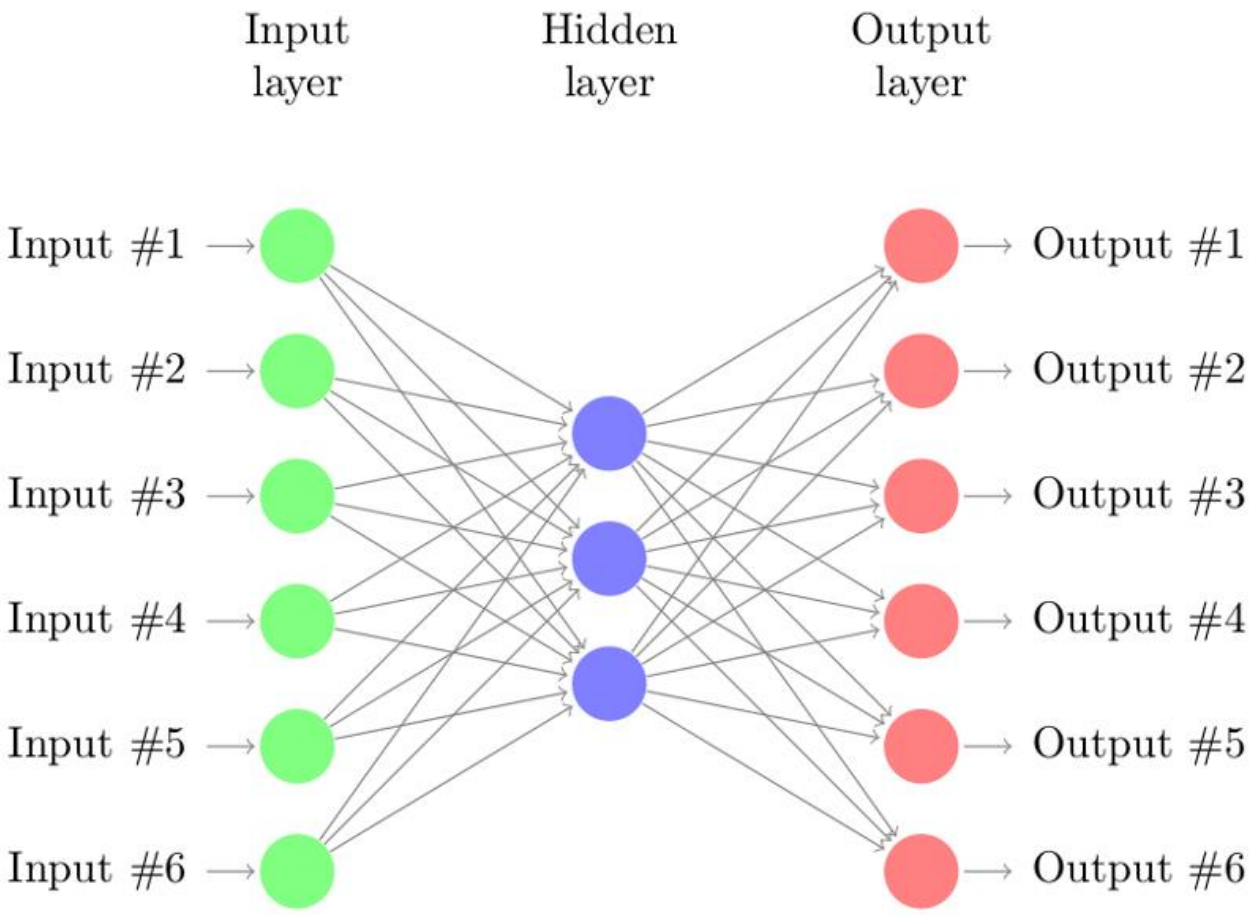
Sparse autoencoders have hidden nodes greater than input nodes. They can still discover important features from the data. A generic sparse autoencoder is visualized where the obscurity of a node corresponds with the level of activation. Sparsity constraint is introduced on the hidden layer. This is to prevent output layer copy input data. Sparsity may be obtained by additional terms in the loss function during the training process, either by comparing the probability distribution of the hidden unit activations with some low desired value,or by manually zeroing all but the strongest hidden unit activations. Some of the most powerful AIs in the 2010s involved sparse autoencoders stacked inside of deep neural networks.

Advantages-

- Sparse autoencoders have a sparsity penalty, a value close to zero but not exactly zero. Sparsity penalty is applied on the hidden layer in addition to the reconstruction error. This prevents overfitting.
- They take the highest activation values in the hidden layer and zero out the rest of the hidden nodes. This prevents autoencoders to use all of the hidden nodes at a time and forcing only a reduced number of hidden nodes to be used.

Drawbacks-

- For it to be working, it's essential that the individual nodes of a trained model which activate are data dependent, and that different inputs will result in activations of different nodes through the network.

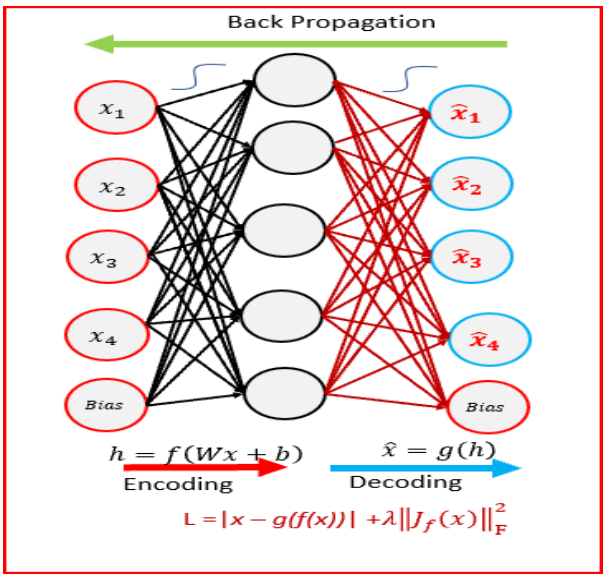


1.2.3) Contractive Autoencoder

The objective of a contractive autoencoder is to have a robust learned representation which is less sensitive to small variation in the data. Robustness of the representation for the data is done by applying a penalty term to the loss function. Contractive autoencoder is another regularization technique just like sparse and denoising autoencoders. However, this regularizer corresponds to the Frobenius norm of the Jacobian matrix of the encoder activations with respect to the input. Frobenius norm of the Jacobian matrix for the hidden layer is calculated with respect to input and it is basically the sum of square of all elements.

Advantages-

- Contractive autoencoder is a better choice than denoising autoencoder to learn useful feature extraction.
- This model learns an encoding in which similar inputs have similar encodings. Hence, we're forcing the model to learn how to contract a neighborhood of inputs into a smaller neighborhood of outputs.



1.3) Representational Power, Layer Size and Depth

A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions.

Layer size refers to the number of neurons in each layer of the network. A larger layer size allows the network to learn more complex representations. However, it is important to note that increasing the layer size too much can lead to overfitting.

Layer depth refers to the number of layers in the network. A deeper network can learn more complex representations than a shallower network. However, deeper networks are also more difficult to train and can be more prone to overfitting.

In general, a larger and deeper autoencoder will have more representational power. However, it is important to choose the appropriate network size and depth for the specific task at hand. If the network is too large or too deep, it may overfit the training data and fail to generalize to new data.

1.4) Stochastic Encoders and Decoders

Stochastic Encoders and Decoders are a type of autoencoder that uses stochasticity (randomness) in the encoding and decoding process. This can be achieved by using dropout, variational inference, or other techniques. Stochastic encoders and decoders have several advantages over deterministic autoencoders, including:

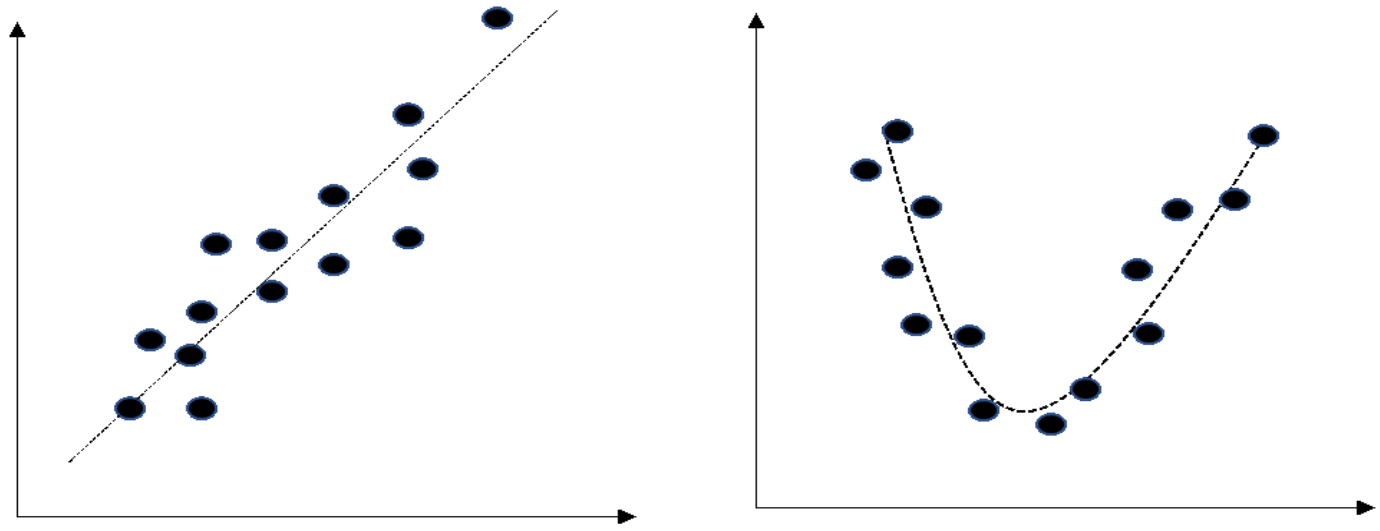
- **Improved generalization**: Stochasticity helps to prevent overfitting by regularizing the model and making it more robust to noise in the training data.
- **More informative representations**: Stochastic encoders and decoders can learn more informative representations of the data by capturing the uncertainty in the data.
- **More efficient training**: Stochastic encoders and decoders can be trained more efficiently than deterministic autoencoders, as they do not need to compute the full posterior distribution of the latent variables.

Stochastic encoders and decoders have been used for a variety of tasks, including:

- **Image denoising**: Stochastic encoders and decoders can be used to denoise images by learning to reconstruct noisy versions of the input data.
- **Inpainting**: Stochastic encoders and decoders can be used to inpaint missing pixels in images by learning to reconstruct the images from partial information.
- **Super-resolution**: Stochastic encoders and decoders can be used to super-resolve images by learning to upsample low-resolution images to high-resolution images.
- **Generative adversarial networks (GANs):** Stochastic encoders and decoders are often used as the generator component in GANs.

1.6) Learning Manifolds with Autoencoders

Manifold learning is an approach in machine learning that assumes that data lies on a manifold of a much lower dimension. These manifolds can be linear or non-linear. Thus, the area tries to project the data from high-dimension space to a low dimension. For example, **principle component analysis** (**PCA**) is an example of linear manifold learning whereas an autoencoder is a **non-linear dimensionality reduction** (**NDR**) with the ability to learn non-linear manifolds in low dimensions. A comparison of linear and non-linear manifold learning is shown in the following figure:



a) Linear manifold

b) Non-linear manifold

As you can see from graph **a)**, the data is residing at a linear manifold, whereas in graph graph **b)**, the data is residing on a second-order non-linear manifold.

All autoencoder training procedures involve a compromise between two forces:

1. Learning a representation h of a training example x such that x can be approximately recovered from h through a decoder. The fact that x is drawn from the training data is crucial, because it means the autoencoder need not successfully reconstruct inputs that are not probable under the data generating distribution.

2. Satisfying the constraint or regularization penalty. This can be an architectural constraint that limits the capacity of the autoencoder, or it can be a regularization term added to the reconstruction cost. These techniques generally prefer solutions that are less sensitive to the input.

1.8) Predictive Sparse Decomposition

Predictive Sparse Decomposition (PSD) is a type of sparse coding algorithm that uses a parametric encoder to predict the sparse representation of the input data. This allows PSD to be used for out-of-sample extension, i.e., to predict the sparse representation of new data that was not seen during training.

PSD works by first training a dictionary of overcomplete basis functions using a standard sparse coding algorithm, such as K-SVD. Once the dictionary is trained, PSD trains a parametric encoder to predict the sparse representation of the input data. The encoder is trained to minimize the reconstruction error between the original input data and the reconstructed data from the sparse representation.

Once the encoder is trained, PSD can be used to predict the sparse representation of new data by simply feeding the new data to the encoder. The encoder will output the predicted sparse representation, which can then be used for a variety of tasks, such as classification, regression, or anomaly detection.

PSD has several advantages over other sparse coding algorithms:

- **Out-of-sample extension:** PSD can be used to predict the sparse representation of new data that was not seen during training. This is not possible with standard sparse coding algorithms, which require the input data to be known in order to compute the sparse representation.
- **Efficient inference:** PSD can be used to efficiently predict the sparse representation of new data. This is because PSD uses a parametric encoder, which is much faster to compute than the iterative algorithms used by standard sparse coding algorithms.
- **Improved generalization:** PSD has been shown to improve generalization performance on a variety of tasks, such as classification and regression. This is because PSD learns to predict the sparse representation of the data, rather than simply memorizing the training data.

1.9) Applications of Autoencoders

Autoencoders have been successfully applied to dimensionality reduction and information retrieval.

Lower dimensional representations can improve performance on many tasks, such as classification.

Information retrieval: the task of finding entries in a database that resemble a query entry. Benefit:

- Usual benefits from dimensionality reduction that other tasks do
- search can be extremely efficient in certain kind of low dimensional spaces. Semantic hashing: train the D reduction algorithm to produce a code that is low-dimensional and binary, then we can store all database entries in a hash table that maps binary code vector to entries. We cam also search over slightly less similar entries efficiently, just by flipping individual bits from the encoding of the query.

Semantic hashing is applied to both textual input and image.

Chapter-II: Deep Generative Models

2.1) Boltzmann Machines

Boltzman Machine definition:

- Over d-D binary random vector $x \in \{0,1\}^d$
- Energy based model. $P(x) = \frac{exp(-E(x))}{Z}$
- $E(x) = -x^T U x - b^T x$, where U is the weight matrix of model parameters and b is the vector of bias parameters.

This senario is certainly viable, it does limit the kind of interactions between the observed variables to those described by the weight matrix. Specifically, it means that the probability of one unit being on is given by a linear model (logistic regression) from the values of the other units.

A Boltzmann machine with hidden units is no longer limited to modeling linear relationship between variables. Instead, the Boltzmann machine becomes a universal approximator of probability mass functions over discrete variables.

Formally, we decompose the unites x into 2 subsets:

- Visible units v
- Latent / Hidden units h

Then the energy function become:

$$E(v,h) = -v^T R v - v^T W h - h^T S h - b^T v - c^T h$$

Learning algorithms for Boltzmann Machines are usually based on maximum likehood. All Boltzmann machines have an intractable partition function, so the maximum likehood gradient must be approximated.

Interesting property of Boltzmann machine when trained with learning fules based one maximum likehood is that update for particular weight connecting 2 unites depends only on the statistics of those 2 units, collected under different distribution $p_{model}(v)$ and $p_{data}(v)p_{model}(h|v)$. The rest of the network participates in shaping those statistics, but the weight can be updated without knowing anything about the rest of the network or how those statistics were produced. This means that the learning rule is local.

2.2) Restricted Boltzmann Machines (RBMs)

A restricted term means that we are not permitted to connect two types of layers that are of the same type to one another. In other words, the two hidden layers or input layers of neurons are unable to form connections with one another. However, there may be connections between the apparent and hidden layers.

Since there is no output layer in our machine, it is unclear how we will detect, modify the weights, and determine whether or not our prediction was right. One response fits all the questions: Restricted Boltzmann Machine.

Features of Restricted Boltzmann Machine

Some key characteristics of the Boltzmann machine are:

- There are no connections between the layers.
- They employ symmetric and recurring structures.

- It is an algorithm for unsupervised learning, meaning that it draws conclusions from the input data without labeled replies.
- In their learning process, RBMs attempt to link low energy states with high probability ones and vice versa.

Working of RBM

A low-level feature from a learning target item in the dataset is used by each visible node. The hidden layer's node 1 multiplies x by weight and adds it to a bias. These two procedures' outcomes are fed into an activation function, which, given an input of x, creates the output of the node, or the signal strength traveling through it.

Let's now examine how many inputs would mix at a single hidden node. The output of the node is created by multiplying each x by a distinct weight, summing the products, adding the sum to a bias, and then passing the final result once again via an activation function.

Each input x is multiplied by its corresponding weight w at each buried node. In other words, a single input x would have three weights in this situation, totaling 12 weights (4 input nodes x 3 hidden nodes). The weights between the two layers will always create a matrix with input nodes in the rows and output nodes in the columns.

The four inputs are sent to each hidden node, multiplied by each weight. Each hidden node receives one output as a result of the activation algorithm after the sum of these products is once more added to a bias (which compels at least some activations to occur).

Now that you have a basic understanding of how the Restricted Boltzmann Machine operates, let's move and examine the procedures for RBM training.

Training of RBM

Two methods - Gibbs Sampling and Contrastive Divergence are used to train RBM.

When direct sampling is challenging, the Markov chain Monte Carlo method known as Gibbs sampling is used to get a series of observations that are roughly drawn from a given multivariate probability distribution.

The prediction is the hidden value by h and $p(h|v)$ if the input is represented by v. $P(v|h)$ is utilized for the regenerated input values' prediction when the hidden values are known. Let's say that after k rounds, v k is acquired from input value v 0 after this process has been performed k times.

In order to approximate the graph slope, a graphical slope showing the relationship between a network's errors and its weights is called the gradient in Contrastive Divergence. Contrastive Divergence is a rough Maximum-Likelihood learning approach and is employed when we need to approximate the learning gradient of the algorithm and choose which direction to go in because we cannot directly evaluate a set of probabilities or a function.

Weights are updated on CD. The gradient is first determined from the reconstructed input, and the old weights are updated by adding the delta.

2.3) **Deep Belief Networks (DBNs)**

Consider stacking numerous RBMs so that the outputs of the first RBM serve as the input for the second RBM, and so forth. Deep Belief Networks are the name given to these networks. Each layer's connections are undirected (as each layer is an RBM). Those between the strata are simultaneously directed (except for the top two layers – whose connections are undirected). The DBNs can be trained in two different ways:

- Greedy Layer-wise Training Algorithm: RBMs are trained using a greedy layer-by-layer training algorithm. The orientation between the DBN layers is established as soon as the individual RBMs have been trained (i.e., the parameters, weights, and biases, have been defined).
- Wake-sleep Algorithm: The DBN is trained from the bottom up using a wake-sleep algorithm (connections going up indicate wake), and then from the bottom up using connections indicating sleep.

In order to ensure that the layer connections only work downwards, we stack the RBMs, train them, and then do so (except for the top two layers).

Chapter-1:Machine Vision

1.1.Convolutional Neural Networks

a) The Two-Dimensional Structure of Visual Imagery

b) Computational Complexity

c) Convolutional Layers

c) Multiple Filters

d) A Convolutional Example

e) Convolutional Filter Hyperparameters

Answer:

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

a) The Two-Dimensional Structure of Visual Imagery

Visual imagery is inherently two-dimensional, with each pixel representing a specific location and intensity value. This two-dimensional structure plays a crucial role in how we perceive and interpret visual information. Convolutional neural networks (CNNs) are specifically designed to exploit this structure, allowing them to effectively process and analyze visual data.

b) Computational Complexity

Traditional neural networks, also known as fully connected networks, connect every neuron in one layer to every neuron in the next layer. This leads to a significant increase in computational complexity as the network grows larger. In contrast, CNNs utilize a convolutional operation that involves sliding a small filter or kernel over the input data. This operation significantly reduces the number of connections, making CNNs more computationally efficient for processing visual data.

c) Convolutional Layers

Convolutional layers are the core building blocks of CNNs. They consist of a set of learnable filters or kernels that are applied to the input data. As the filters slide across the input, they extract relevant features, such as edges, shapes, and colors. These extracted features are then passed to subsequent layers for further processing and analysis.

d) Multiple Filters

CNNs can employ multiple filters in each convolutional layer, each capturing different aspects of the input data. For instance, one filter might detect horizontal edges, while another might detect vertical edges. By using multiple filters, CNNs can learn a more comprehensive representation of the input data.

e) A Convolutional Example

Consider a simple image of a cat. A CNN might use a filter that detects horizontal edges to identify the cat's whiskers. Another filter might detect vertical edges to identify the cat's eyes and mouth. By combining the outputs of these filters, the CNN can build a more complete understanding of the cat's features and ultimately classify the image as a cat.

f) Convolutional Filter Hyperparameters

Convolutional filters are characterized by several hyperparameters that influence their behavior. These include:

- **Filter size:** The size of the filter determines the scope of the features it detects. Larger filters capture larger-scale features, while smaller filters capture finer details.
- **Stride:** The stride determines how far the filter moves across the input data with each application. Larger strides lead to more efficient computation but may overlook smaller details.
- **Padding:** Padding adds zeros to the edges of the input data, allowing the filter to extract features that extend beyond the original boundaries.
- **Activation function:** The activation function introduces non-linearity into the network, allowing it to learn more complex patterns. Common activation functions for CNNs include ReLU (Rectified Linear Unit) and Leaky ReLU.

1.2.Pooling Layers

1.3.LeNet-5 in Keras

1.4.AlexNet and VGGNet in Keras

Answer:

1.2. Pooling Layers

Pooling layers are another essential component of CNNs. Their primary purpose is to reduce the spatial dimensions of the feature maps produced by convolutional layers. This reduction in dimensionality helps to control the computational complexity of the network and prevent overfitting.

There are two main types of pooling layers: max pooling and average pooling.

Max pooling:

Max pooling selects the maximum value within each pooling region, effectively reducing the spatial dimensions by a factor of the pooling size. For instance, a 2x2 max pooling layer would reduce the height and width of the feature map by half.

Average pooling:

Average pooling computes the average value within each pooling region, also reducing the spatial dimensions by a factor of the pooling size. This type of pooling tends to be more robust to outliers and noise compared to max pooling.

Pooling layers are typically applied after convolutional layers to reduce the feature map size before passing it on to subsequent layers. The choice of pooling type (max pooling or average pooling) and pooling size depends on the specific task and dataset.

1.3. LeNet-5 in Keras

LeNet-5 is a seminal convolutional neural network architecture developed by Yann LeCun in 1998. It was designed for handwritten digit recognition and achieved remarkable performance on the MNIST dataset. LeNet-5 is considered a groundbreaking architecture that set the foundation for modern CNNs.

Here's a simplified overview of the LeNet-5 architecture:

- **Input layer:** Receives a 28x28 grayscale image representing a handwritten digit.
- **Convolutional layer 1:** Applies 6 5x5 filters to the input image, producing 6 feature maps.
- **Pooling layer 1:** Applies a 2x2 max pooling operation to each feature map, reducing the size by half.
- **Convolutional layer 2:** Applies 16 5x5 filters to the pooled feature maps, producing 16 feature maps.
- **Pooling layer 2:** Applies a 2x2 max pooling operation to each feature map, reducing the size by half.

- **Fully connected layer 1:** Flattens the pooled feature maps into a 120-dimensional vector.
- **Fully connected layer 2:** Reduces the 120-dimensional vector to 84 dimensions.
- **Output layer:** Produces a 10-dimensional vector representing the probabilities of the input image being each digit (0-9).

Keras provides a built-in implementation of LeNet-5, allowing you to easily train and evaluate this architecture for handwritten digit recognition.

1.4. AlexNet and VGGNet in Keras

AlexNet and VGGNet are two more advanced CNN architectures that gained popularity in the early 2010s. These architectures achieved state-of-the-art performance on image classification tasks, demonstrating the power of CNNs in computer vision.

AlexNet:

AlexNet was the winner of the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), achieving a significant improvement in image classification accuracy compared to previous methods. It introduced deeper architectures with more convolutional and pooling layers, demonstrating the potential of deeper CNNs.

VGGNet:

VGGNet is a family of CNN architectures that explore the effect of increasing the depth of the network. It introduced the concept of using very small 3x3 filters stacked in deeper layers, achieving competitive performance with AlexNet while using fewer parameters.

Keras provides built-in implementations of both AlexNet and VGGNet, allowing you to easily train and evaluate these architectures for image classification tasks.

1.5.Residual Networks

a) Vanishing Gradients: The Bête Noire of Deep CNNs

b) Residual Connections

c) ResNet

Answer:

Residual Network: In order to solve the problem of the vanishing/exploding gradient, this architecture introduced the concept called Residual Blocks. In this network, we use a technique called ***skip connections***. The skip connection connects activations of a layer to further layers by skipping some layers in between. This forms a residual block. Resnets are made by stacking these residual blocks together. The approach behind this network is instead of layers learning the underlying mapping, we allow the network to fit the residual mapping. So, instead of say H(x), initial mapping, let the network fit,

F(x) := H(x) - x which gives *H(x) := F(x) + x.*

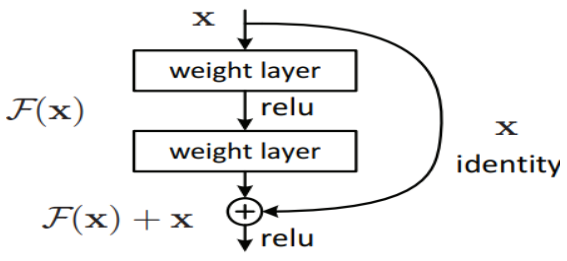


Fig: Skip (Shortcut) connection

The advantage of adding this type of skip connection is that if any layer hurt the performance of architecture then it will be skipped by regularization. So, this results in training a very deep neural network without the problems caused by vanishing/exploding gradient. The authors of the paper experimented on 100-1000 layers of the CIFAR-10 dataset.

a) Vanishing Gradients: The Bête Noire of Deep CNNs

As CNNs become deeper, a phenomenon known as vanishing gradients arises. This means that the gradients of the loss function with respect to the weights in lower layers become very small, making it difficult for the optimization algorithm to update these weights effectively.

This problem is particularly acute in very deep CNNs, where the gradients can become so small that they are effectively zero, preventing any meaningful update to the weights. This makes it difficult for these deep networks to learn and improve their performance.

b) Residual Connections

Residual networks (ResNets) were introduced in 2015 by Kaiming He et al. to address the problem of vanishing gradients in deep CNNs. ResNets introduce a novel architectural design that allows for efficient information flow through the network, even when it is very deep.

The key idea behind ResNets is to introduce shortcut connections that bypass one or more layers in the network. These shortcut connections allow the gradients to flow directly from the input of a block to its output, without having to pass through all of the intermediate layers. This helps to prevent the gradients from vanishing, allowing the network to learn more effectively.

c) ResNet

The ResNet architecture consists of a series of residual blocks, each containing a convolutional layer followed by a shortcut connection. The shortcut connection typically adds the input of the block to the output of the convolutional layer, before applying an activation function. This design allows for efficient information flow through the network, even when it is very deep.

ResNets have achieved remarkable performance in a wide range of image recognition tasks, including image classification, object detection, and image segmentation. They have become the de-facto standard for deep CNN architectures and have been widely adopted in the field of computer vision.

1.6.Applications of Machine Vision

a) Object Detection

b) Image Segmentation

c) Transfer Learning

d) Capsule Networks

Answer:

Machine vision has revolutionized various industries by automating tasks that were previously performed manually. Its applications span a wide range, from manufacturing and robotics to healthcare and agriculture. Here are some notable examples of machine vision applications:

a) Object Detection

Object detection involves identifying and locating objects within images or videos. This capability is crucial for tasks such as:

- **Traffic monitoring:** Machine vision systems can detect and track vehicles in traffic footage, enabling real-time traffic analysis and accident prevention.

- **Security and surveillance:** Object detection algorithms can identify and track people or objects in surveillance footage, enhancing security measures and preventing unauthorized access.
- **Retail inventory management:** Machine vision systems can automate product counting and tracking on shelves, optimizing inventory management and preventing stockouts.

b) Image Segmentation

Image segmentation involves dividing an image into meaningful regions or segments. This technique is essential for tasks such:

- **Medical image analysis:** Machine vision algorithms can segment tissues and organs in medical images, aiding in diagnosing diseases and planning treatments.
- **Satellite image analysis:** Image segmentation can identify and classify land cover types from satellite imagery, supporting environmental monitoring and resource management.
- **Autonomous driving:** Image segmentation helps autonomous vehicles distinguish between roads, pedestrians, and other objects, enabling safe and reliable navigation.

c) Transfer Learning

Transfer learning involves utilizing a pre-trained machine learning model for a new task. This approach is particularly valuable for machine vision applications, as training large-scale deep learning models can be computationally expensive and time-consuming.

Transfer learning allows researchers to leverage existing models, such as those trained on large image datasets like ImageNet, and adapt them to specific applications. This can significantly reduce the training time and computational resources required for developing new machine vision systems.

d) Capsule Networks

Capsule networks, introduced by Geoffrey Hinton in 2017, represent a recent advancement in machine vision. They aim to address some of the limitations of traditional convolutional neural networks (CNNs), particularly in capturing spatial relationships and object hierarchies.

Capsule networks group neurons into capsules, where each capsule represents a higher-level concept or feature in the image. This hierarchical structure allows capsule networks to learn more robust representations of objects and their relationships.

While capsule networks are still under development, they hold promise for improving the performance of machine vision tasks in areas such as object detection, image segmentation, and image understanding.

Application	Description	Example Tasks
Object detection	Identifying and locating objects in images or videos	Traffic monitoring, security surveillance, retail inventory management
Image segmentation	Dividing an image into meaningful regions or segments	Medical image analysis, satellite image analysis, autonomous driving
Transfer learning	Utilizing a pre-trained machine learning model for a new task	Adapting existing image classification models to specific applications, such as product recognition or defect detection
Capsule networks	A novel neural network architecture that aims to address limitations of traditional CNNs	Potential improvements in object detection, image segmentation, and image understanding

Chapter-II: Natural Language processing

2.1. Preprocessing Natural Language Data

a) Tokenization

b) Converting All Characters to Lowercase

c) Removing Stop Words and Punctuation

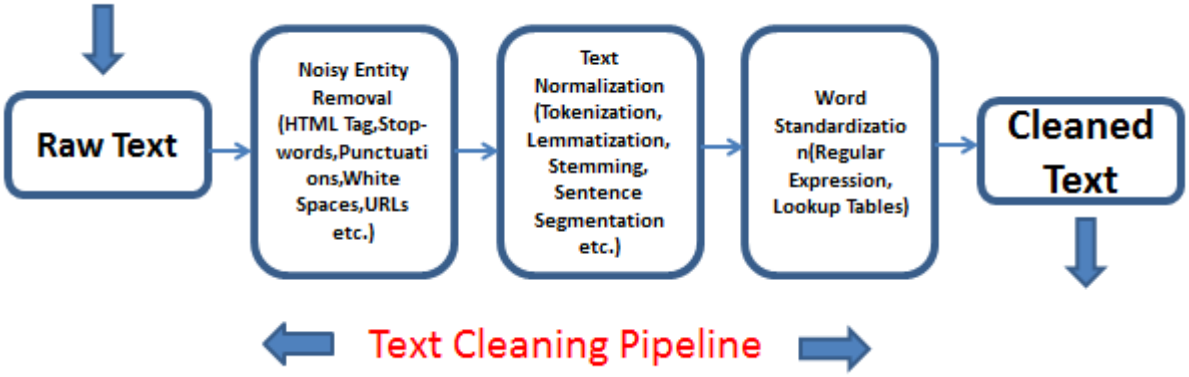
d) Stemming

e) Handling n-grams

f) Preprocessing the Full Corpus

Answer:

Natural language data, consisting of text and speech, is often unstructured and noisy, making it directly unsuitable for machine learning algorithms. Preprocessing natural language data involves cleaning, transforming, and structuring the data to make it suitable for analysis and modeling.



Here are some essential steps in preprocessing natural language data:

a) Tokenization

Tokenization is the process of breaking down a text into individual units, called tokens. Tokens can be words, punctuation marks, or even individual characters. Tokenization is the first step in many natural language processing tasks, as it allows the algorithm to understand the basic structure of the text.

b) Converting All Characters to Lowercase

Converting all characters to lowercase is a simple but effective way to reduce the dimensionality of the data. This is because many words have different forms depending on their capitalization, such as "dog" and "Dog". By converting all characters to lowercase, we can treat these words as equivalent, reducing the number of unique tokens.

c) Removing Stop Words and Punctuation

Stop words are common words that don't carry much meaning, such as "the", "a", and "an". Removing stop words can reduce the noise in the data and improve the performance of machine learning algorithms. Punctuation marks can also be removed, as they often don't contribute to the meaning of the text.

d) Stemming and Lemmatization

Stemming and lemmatization are techniques for reducing words to their base forms. Stemming is a more aggressive process that removes affixes (prefixes and suffixes) from words, while lemmatization takes into account the context of the word to identify its root form.

For example, stemming the words "cats", "catches", and "catching" would all result in the stem "cat". Lemmatization, on the other hand, would identify "cat" as the lemma for all three words.

e) Handling n-grams

N-grams are sequences of n consecutive tokens. For example, the bigrams (two-word sequences) for the phrase "natural language processing" would be "natural language", "language processing", and "processing".

N-grams can be useful for capturing local patterns in the data, such as collocations (words that frequently appear together). However, it's important to choose the appropriate value of n, as too large of a value can lead to a large number of n-grams and increase the dimensionality of the data.

f) Preprocessing the Full Corpus

Preprocessing the full corpus, or the entire collection of text data, involves applying the above steps to all the text documents in the corpus. This ensures that the data is consistent and ready for further analysis or modeling.

2.2. Creating Word Embeddings with word2vec

a) The Essential Theory Behind word2vec

b) Evaluating Word Vectors

c) Running word2vec

d) Plotting Word Vectors

Answer:

a) The Essential Theory Behind word2vec

word2vec is a family of algorithms and models for producing word embeddings, which are vector representations of words. These vector representations capture the semantic and syntactic relationships between words, allowing machines to understand the meaning and context of words in natural language.

There are two main word2vec algorithms:

- **Continuous Bag-of-Words (CBOW):** CBOW predicts a target word from its surrounding context words. It does this by training a neural network that takes a sequence of context words as input and outputs a vector representation of the target word.
- **Skip-gram:** Skip-gram predicts the surrounding context words from a target word. It does this by training a neural network that takes a vector representation of a target word as input and outputs a probability distribution over all possible context words.

Both CBOW and skip-gram effectively learn word embeddings that capture the relationships between words in a corpus of text. These word embeddings can then be used for various natural language processing tasks, such as:

- **Word similarity:** Measuring the similarity between words based on their vector representations.
- **Semantic analogy:** Solving analogies like "King is to Queen as Man is to ?" using word embeddings.
- **Sentiment analysis:** Classifying the sentiment of text (positive, negative, neutral) based on word embeddings.

b) Evaluating Word Vectors

Evaluating the quality of word embeddings is crucial for ensuring their effectiveness in downstream tasks. Several methods can be used to evaluate word embeddings:

- **Lexical similarity tasks:** Measuring how well word embeddings capture word similarity, such as using the WordSim353 dataset.

- **Analogy tasks:** Measuring how well word embeddings can solve word analogies, such as using the Stanford Analogy Test.
- **Intrinsic evaluation metrics:** Measuring the internal consistency of word embeddings, such as using the cosine similarity between words with similar meanings.

c) Running word2vec

There are various tools and libraries available for implementing word2vec algorithms, such as Gensim and TensorFlow. These tools provide user-friendly interfaces for training and using word2vec models.

The general process of running word2vec involves:

1. **Preparing the corpus:** Cleaning and preprocessing the text data to remove noise and inconsistencies.
2. **Training the word2vec model:** Selecting the appropriate algorithm (CBOW or skip-gram), setting hyperparameters (embedding size, window size), and training the model on the preprocessed text data.
3. **Evaluating the word embeddings:** Using evaluation metrics to assess the quality of the learned word vectors.
4. **Saving and using the word embeddings:** Saving the word embeddings for later use in downstream tasks.

d) Plotting Word Vectors

Visualizing word embeddings can provide insights into their relationships and semantic properties. Two common techniques for plotting word vectors are:

- **2D projection:** Reducing the dimensionality of word embeddings to two dimensions and plotting them in a scatter plot. This allows for visualizing the relative positions of words in the vector space.
- **TSNE:** Using t-distributed stochastic neighbor embedding (TSNE) to project high-dimensional word embeddings into a lower-dimensional space while preserving local and global relationships. This produces a more nuanced visualization of word relationships.

By plotting word vectors, we can observe how words with similar meanings cluster together, identify outliers, and gain insights into the semantic structure of the language.

2.3. The Area under the ROC Curve

a) The Confusion Matrix

b) Calculating the ROC AUC Metric

Answer:

a) The Confusion Matrix

A confusion matrix is a table that summarizes the performance of a binary classifier over a set of test data. It is a tabular representation of the actual and predicted classifications for each class in the dataset.

Actual\Predicted	Positive	Negative
Positive	True Positives (TP)	False Negatives (FN)
Negative	False Positives (FP)	True Negatives (TN)

- **True Positives (TP):** Instances correctly classified as positive.
- **False Positives (FP):** Instances incorrectly classified as positive.
- **False Negatives (FN):** Instances incorrectly classified as negative.

- **True Negatives (TN):** Instances correctly classified as negative.

b) Calculating the ROC AUC Metric

The receiver operating characteristic (ROC) curve is a graphical plot that illustrates the performance of a binary classifier at varying threshold settings. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) at each threshold setting.

The area under the ROC curve (AUC) is a single scalar value that measures the overall performance of a binary classifier. It is a measure of how well the classifier can distinguish between positive and negative cases.

TP / (TP + FN) Probability of classifying a positive case as positive

FP / (FP + TN) Probability of classifying a negative case as positive

A higher AUC indicates a better classifier, as it means that the classifier is better at distinguishing between positive and negative cases. A perfect classifier would have an AUC of 1.0, while a random classifier would have an AUC of 0.5.

Here's the formula for calculating the AUC:

$$AUC = \int TPR(FPR)\, dFPR$$

This formula represents the integral of the TPR curve over the range of FPR values from 0 to 1. In practice, the AUC is often calculated using numerical methods, such as the trapezoidal rule.

2.4.Natural Language Classification with Familiar Networks

a) Loading the IMDb Film Reviews

b) Examining the IMDb Data

c) Standardizing the Length of the Reviews

d) Dense Network

e) Convolutional Networks

Answer:

a) Loading the IMDb Film Reviews

The IMDb dataset for sentiment analysis consists of a collection of movie reviews labeled as either positive or negative. This dataset is a valuable resource for training and evaluating natural language processing models for sentiment classification.

Here's the code snippet for loading the IMDb dataset using Keras:

```
from keras.datasets import imdb

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=10000)
```

This code snippet loads the IMDb dataset, limits the vocabulary to the 10,000 most frequently occurring words, and splits the data into training and testing sets.

b) Examining the IMDb Data

The IMDb dataset contains movie reviews in the form of text sequences. Each review is represented as a list of integers, where each integer corresponds to a unique word in the vocabulary.

Here's an example of a review and its corresponding word indices:

Review: "A very good movie with an excellent performance by the lead actor."

Word Indices: [123, 456, 789, 1011, 2345, 6789, ...]

The task of sentiment classification involves predicting the sentiment (positive or negative) of a given review based on its word indices.

c) Standardizing the Length of the Reviews

Movie reviews can vary significantly in length, which can affect the performance of machine learning models. To standardize the length of the reviews, we can pad shorter reviews with zeros and truncate longer reviews to a fixed length.

Here's an example of padding shorter reviews:

```
max_length = 100  # Maximum length of a review

x_train = pad_sequences(x_train, maxlen=max_length)

x_test = pad_sequences(x_test, maxlen=max_length)
```

This code snippet pads all reviews in the training and testing sets to a maximum length of 100.

d) Dense Network

A dense network, also known as a fully connected network, is a simple and effective architecture for natural language classification. It consists of a series of fully connected layers, where each layer is connected to every neuron in the previous layer.

Here's an example of a dense network for sentiment classification:

```
from keras.models import Sequential

from keras.layers import Dense, Embedding, LSTM

model = Sequential()

model.add(Embedding(10000, 128, input_length=max_length))

model.add(LSTM(64))

model.add(Dense(1, activation='sigmoid'))
```

This model consists of three layers: an embedding layer, an LSTM layer, and a dense output layer. The embedding layer converts word indices into vector representations. The LSTM layer learns long-range dependencies in the text sequences. The dense output layer produces a probability between 0 and 1, indicating the sentiment of the review.

e) Convolutional Networks

Convolutional neural networks (CNNs) are another powerful architecture for natural language processing tasks. CNNs are particularly well-suited for extracting local patterns and features from text data.

Here's an example of a convolutional network for sentiment classification:

```
from keras.models import Sequential

from keras.layers import Conv1D, MaxPooling1D, Dense, Embedding

model = Sequential()

model.add(Embedding(10000, 128, input_length=max_length))

model.add(Conv1D(64, filter_size=3, activation='relu'))

model.add(MaxPooling1D(pool_size=2))
```

model.add(Dense(1, activation='sigmoid'))

This model consists of three layers: an embedding layer, a convolutional layer, and a dense output layer. The convolutional layer applies convolutional filters to extract local patterns from the text sequences. The max pooling layer reduces the dimensionality of the feature maps. The dense output layer produces a probability between 0 and 1, indicating the sentiment of the review.

Both dense networks and convolutional networks can be effective for natural language classification. The choice of architecture depends on the specific task and dataset.

2.5.Networks Designed for Sequential Data

a) Recurrent Neural Networks

b) Long Short-Term Memory Units

c) Bidirectional LSTMs

d) Stacked Recurrent Models

e) Seq2seq and Attention

e) Transfer Learning in NLP

Answer:

2.5. Networks Designed for Sequential Data

a) Recurrent Neural Networks

Recurrent neural networks (RNNs) are a class of neural networks specifically designed for processing sequential data, such as text, speech, and time series data. RNNs are able to capture long-range dependencies in sequential data by maintaining an internal state that is updated as the data is processed.

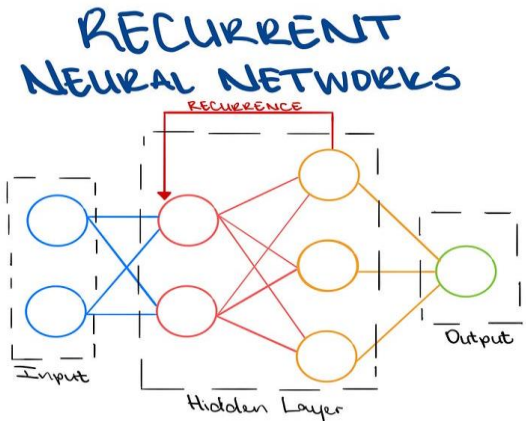


Fig:Recurrent Neural Network (RNN)

The basic unit of an RNN is a recurrent unit, which is a neural network that takes its own state as input in addition to the current input. This allows the recurrent unit to remember information about previous inputs, which is crucial for understanding sequential data.

b) Long Short-Term Memory Units

Long short-term memory (LSTM) units are a type of recurrent unit that are specifically designed to address the vanishing gradient problem, which is a common issue in RNNs that can make it difficult to learn long-range dependencies. LSTM units have a complex gating mechanism that allows them to selectively remember and forget information, making them more effective at learning long-range dependencies than traditional RNNs.

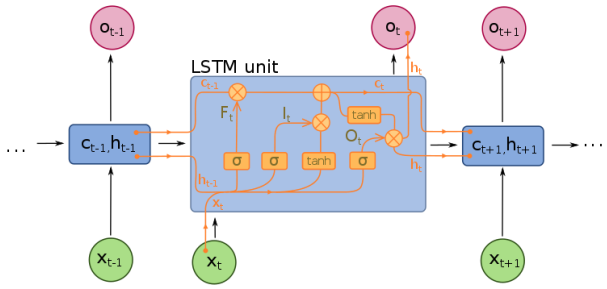


Fig:Long ShortTerm Memory (LSTM) Unit

LSTM units have become the dominant architecture for RNNs and are widely used in a variety of natural language processing tasks, such as machine translation, sentiment analysis, and speech recognition.

c) Bidirectional LSTMs

Bidirectional LSTMs (Bi-LSTMs) are a type of LSTM that takes advantage of the fact that natural language is often processed in two directions, from left to right and from right to left. Bi-LSTMs process the input sequence in both directions and then combine the two representations to produce a more comprehensive understanding of the input.

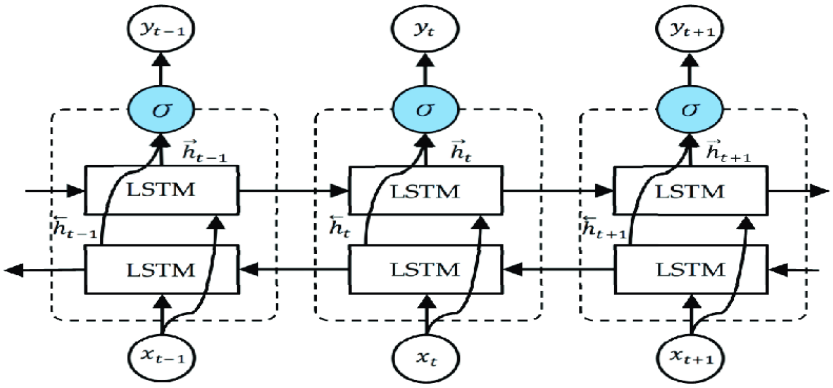


Fig:Bidirectional LSTM (BiLSTM) Unit

Bi-LSTMs have been shown to be particularly effective for tasks that require understanding both the context and the order of words in a sentence, such as machine translation and named entity recognition.

d) Stacked Recurrent Models

Stacked recurrent models are a type of RNN that consists of multiple layers of recurrent units stacked on top of each other. Each layer of recurrent units takes the output of the previous layer as input, allowing the model to capture more complex patterns in the data.

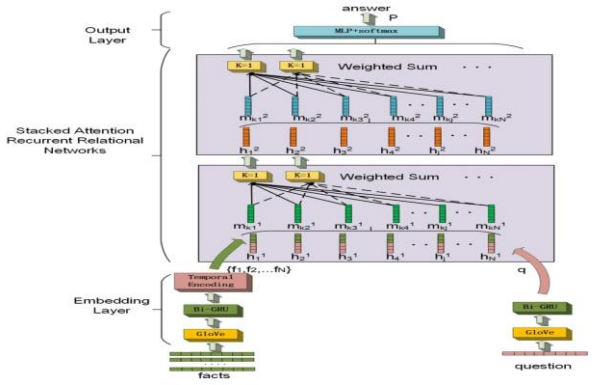


Fig:Stacked Recurrent Model

Stacked recurrent models have been shown to be effective for tasks that require a deep understanding of the input sequence, such as machine translation and speech recognition.

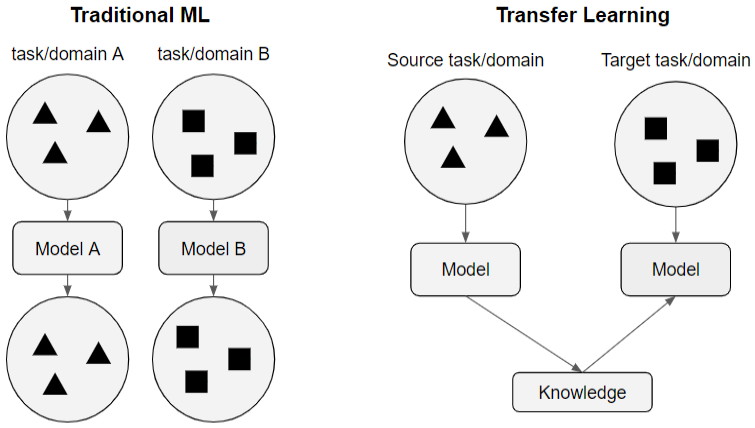
e) Seq2seq and Attention

Seq2seq models are a type of RNN that is specifically designed for tasks that involve translating sequences from one language to another. Seq2seq models typically consist of an encoder-decoder architecture, where the encoder encodes the input sequence into a vector representation and the decoder generates the output sequence based on the vector representation.

Attention is a mechanism that allows the decoder to focus on the most relevant parts of the input sequence when generating the output sequence. This can improve the accuracy of the model, especially for long input sequences.

e) Transfer Learning in NLP

Transfer learning is a technique that involves using a pre-trained model on a new task. This can be particularly useful in natural language processing, where training large neural networks from scratch can be computationally expensive and time-consuming.



Transfer Learning in NLP

There are several ways to apply transfer learning in NLP. One common approach is to use the output of a pre-trained model as input to a new model. Another approach is to fine-tune the parameters of a pre-trained model on the new task.

Transfer learning has been shown to be effective for a variety of NLP tasks, including machine translation, sentiment analysis, and text classification.

Chapter-III: Generative Adversial Networks

3.1.Essential GAN Theory

3.3.The Quick, Draw! Dataset

3.4.The Discriminator Network

3.5.The Generator Network

3.6.The Adversarial Network

3.7.GAN Training

Answer:

A Generative Adversarial Network (GAN) is a deep learning architecture that consists of two neural networks competing against each other in a zero-sum game framework. The goal of GANs is to generate new, synthetic data that resembles some known data distribution.

3.1. Essential GAN Theory

Generative adversarial networks (GANs) are a class of machine learning models that are able to generate new data that is similar to real data. GANs are composed of two main components: a generator and a discriminator.

The Generator:

The generator is responsible for generating new data. It takes in a random noise vector as input and produces a synthetic data sample as output. The goal of the generator is to produce data that is so realistic that it can fool the discriminator into believing it is real.

The Discriminator:

The discriminator is responsible for distinguishing between real and fake data. It takes in a data sample as input and outputs a probability that the sample is real. The goal of the discriminator is to correctly classify real data as real and fake data as fake.

The generator and discriminator are trained in an adversarial manner. The generator tries to improve its ability to generate realistic data, while the discriminator tries to improve its ability to distinguish between real and fake data. This adversarial training process forces both the generator and discriminator to improve their performance, leading to the generation of increasingly realistic data.

3.3. The Quick, Draw! Dataset

The Quick, Draw! dataset is a large collection of hand-drawn sketches paired with corresponding labels. The dataset was created by Google and is freely available for research purposes.

The Quick, Draw! dataset is a valuable resource for training GANs because it provides a large amount of real data that can be used to train the discriminator. Additionally, the dataset is labeled, which allows the training process to be supervised.

3.4. The Discriminator Network

The discriminator network is a convolutional neural network (CNN) that takes in a sketch as input and outputs a probability that the sketch is real. The CNN architecture is well-suited for this task because it is able to extract local features from the sketch, which can be used to distinguish between real and fake sketches.

The discriminator network consists of several convolutional layers followed by a fully connected layer. The convolutional layers extract features from the sketch, while the fully connected layer outputs the probability that the sketch is real.

3.5. The Generator Network

The generator network is a recurrent neural network (RNN) that takes in a random noise vector as input and produces a sketch as output. The RNN architecture is well-suited for this task because it is able to generate sequences of data, which is necessary for generating sketches.

The generator network consists of an embedding layer, an LSTM layer, and a decoder layer. The embedding layer converts the random noise vector into a vector representation. The LSTM layer processes the vector representation and generates a sequence of hidden states. The decoder layer takes the hidden states as input and produces a sketch as output.

3.6. The Adversarial Network

The adversarial network is the combination of the generator and discriminator networks. The adversarial network is trained in an adversarial manner, where the generator tries to fool the discriminator into believing its sketches are real, while the discriminator tries to correctly classify real sketches as real and fake sketches as fake.

The adversarial training process forces both the generator and discriminator to improve their performance. The generator learns to generate more realistic sketches, while the discriminator learns to better distinguish between real and fake sketches.

3.7. GAN Training

GAN training is a complex process that requires careful tuning of hyperparameters. The following are some of the key hyperparameters that need to be tuned:

- **Learning rate:** The learning rate determines how quickly the generator and discriminator networks are updated during training. A high learning rate can lead to unstable training, while a low learning rate can lead to slow training.
- **Batch size:** The batch size determines the number of data samples that are used to update the generator and discriminator networks at each training step. A large batch size can lead to more stable training, while a small batch size can lead to faster training.
- **Regularization:** Regularization techniques can be used to prevent the generator and discriminator networks from overfitting the training data. Common regularization techniques include dropout and L2 regularization.
- **Loss function:** The loss function determines how the performance of the generator and discriminator networks is measured. The most common loss function for GANs is the binary cross-entropy loss.

GAN training can be a challenging process, but it can be very rewarding when successful. GANs have been used to generate a variety of realistic data, including images, videos, and music.

Chapter-IV: Deep Reinforcement Learning

4.1.Essential Theory of Reinforcement Learning

a) The Cart-Pole Game

b) Markov Decision Processes

c) The Optimal Policy

Answer:

a) The Cart-Pole Game

The Cart-Pole game is a classic reinforcement learning task that involves balancing a pole on a cart. The cart can move left or right, and the goal is to keep the pole upright as long as possible. The game is typically played in a simulated environment, where the state of the cart and pole is represented by a vector of numbers.

The Cart-Pole game is a good example of a reinforcement learning problem because it has the following characteristics:

- **The environment is dynamic:** The state of the cart and pole changes over time.
- **The environment is stochastic:** The outcome of an action is not always deterministic.
- **There is a delayed reward:** The agent does not receive a reward immediately after taking an action.

b) Markov Decision Processes

A Markov decision process (MDP) is a mathematical framework for modeling reinforcement learning problems. An MDP is defined by the following elements:

- **A set of states:** The states represent the possible configurations of the environment.
- **A set of actions:** The actions are the choices that the agent can make.
- **A transition function:** The transition function defines how the environment transitions from one state to another given an action.
- **A reward function:** The reward function defines the immediate reward that the agent receives for taking an action.

The goal of reinforcement learning is to find an optimal policy, which is a function that maps states to actions and maximizes the expected cumulative reward.

c) The Optimal Policy

The optimal policy is the policy that maximizes the expected cumulative reward. The expected cumulative reward is the sum of the immediate rewards that the agent expects to receive over an infinite time horizon.

There are two main approaches to finding the optimal policy:

- **Value-based methods:** Value-based methods estimate the value of each state and use this information to select the action that will lead to the highest expected cumulative reward.
- **Policy-based methods:** Policy-based methods directly update the policy using gradient descent or other optimization techniques.

Both value-based and policy-based methods have their own strengths and weaknesses. Value-based methods are often more stable and less prone to overfitting, while policy-based methods can be more efficient and can sometimes find policies that are closer to the optimal policy.

4.2.Essential Theory of Deep Q-Learning Networks

a) Value Functions

b) Q-Value Functions

c) Estimating an Optimal Q-Value

Answer:

a) Value Functions

A value function is a function that maps states to their expected cumulative reward. In other words, the value function estimates how much reward an agent can expect to receive if it starts in a given state and follows an optimal policy.

The value function for a state s is denoted by V(s). The value function can be computed using the following recursive equation:

$V(s) = \max_a [R(s, a) + \gamma \Sigma_{s'} P(s'|s, a) V(s')]$

where:

- R(s, a) is the immediate reward for taking action a in state s.
- γ is the discount factor, which determines how much the agent values future rewards.
- P(s'|s, a) is the probability of transitioning to state s' after taking action a in state s.
- Σ_s' is the sum over all possible next states s'.

b) Q-Value Functions

A Q-value function is a function that maps state-action pairs to their expected cumulative reward. In other words, the Q-value function estimates how much reward an agent can expect to receive if it takes a given action in a given state and then follows an optimal policy.

The Q-value function for a state-action pair (s, a) is denoted by Q(s, a). The Q-value function can be computed using the following recursive equation:

$Q(s, a) = R(s, a) + \gamma \max_{a'} \Sigma_{s'} P(s'|s, a) V(s')$

where:

- R(s, a) is the immediate reward for taking action a in state s.
- γ is the discount factor, which determines how much the agent values future rewards.
- P(s'|s, a) is the probability of transitioning to state s' after taking action a in state s.
- max_a' is the maximum over all possible next actions a'.
- Σ_s' is the sum over all possible next states s'.

c) Estimating an Optimal Q-Value

Deep Q-learning networks (DQNs) are a type of reinforcement learning algorithm that uses a neural network to estimate the Q-value function. DQNs are trained using an experience replay buffer, which stores past experiences of the agent.

The DQN algorithm works as follows:

1. The agent interacts with the environment and collects experiences.
2. The experiences are stored in an experience replay buffer.

3. The neural network is trained on a batch of experiences from the replay buffer.
4. The agent uses the updated neural network to select actions.

The goal of training the neural network is to minimize the following loss function:

$L = \Sigma_{(s,a,r,s')} [Q(s, a) - (r + \gamma \max_{a'} Q(s', a'))]^2$

where:

- (s, a, r, s') is an experience from the replay buffer.
- Q(s, a) is the predicted Q-value for the experience.
- r is the immediate reward for the experience.
- γ is the discount factor.
- max_a' Q(s', a') is the maximum predicted Q-value for the next state in the experience.

DQNs have been shown to be very effective in a variety of reinforcement learning tasks, including Atari games, robotics, and finance.

4.3.Defining a DQN Agent

a) Initialization Parameters

b) Building the Agent's Neural Network Model

c) Remembering Gameplay

d) Training via Memory Replay

e) Selecting an Action to Take

f) Saving and Loading Model Parameters

Answer:

a) Initialization Parameters

Before diving into the agent's architecture and training process, it's crucial to define the essential initialization parameters that govern the agent's behavior and learning process. These parameters include:

- **Discount factor (γ):** This parameter determines the importance of future rewards compared to immediate ones. A higher discount factor emphasizes long-term rewards, while a lower discount factor prioritizes immediate gratification.
- **Exploration rate (ε):** This parameter controls the balance between exploration (trying new actions) and exploitation (sticking to known good actions). A higher exploration rate encourages exploration of new strategies, while a lower exploration rate focuses on exploiting existing knowledge.
- **Memory replay buffer size:** This parameter determines the capacity of the agent's memory, which stores past experiences for training. A larger memory allows for more diverse training data, but it also increases computational cost.
- **Target network update frequency:** This parameter specifies how often the target network, used for generating Q-values during training, is updated from the main network. A higher update frequency ensures the target network remains aligned with the latest learning progress.
- **Batch size:** This parameter defines the number of experiences sampled from the memory replay buffer for each training update. A larger batch size improves training efficiency but may lead to overfitting.

b) Building the Agent's Neural Network Model

The core of a DQN agent is its neural network model, responsible for estimating Q-values. The network architecture typically consists of an input layer, hidden layers, and an output layer. The input layer receives the current state representation of the environment, while the output layer produces Q-values for each possible action.

The choice of activation functions for the hidden layers depends on the specific task and the nature of the input data. Common activation functions include rectified linear units (ReLU) and hyperbolic tangent (tanh).

c) Remembering Gameplay

The agent's memory replay buffer plays a crucial role in DQN's learning process. It stores past experiences, also known as transitions, in the form of tuples containing the current state, the action taken, the immediate reward received, and the next state after the action.

By storing these experiences, the agent can revisit and learn from past mistakes or successful strategies, allowing it to improve its decision-making over time.

d) Training via Memory Replay

The training process of a DQN agent involves sampling batches of experiences from its memory replay buffer. These experiences are used to update the neural network model by minimizing the loss function, which measures the difference between the predicted Q-values and the actual rewards obtained.

The loss function is typically calculated using the Bellman equation, which allows the agent to propagate the value information from future states back to the current state.

e) Selecting an Action to Take

When interacting with the environment, the agent faces the decision of choosing an action to take based on the current state. The agent's policy, guided by the Q-values estimated by the neural network, determines how it balances exploration and exploitation.

With a high exploration rate, the agent may select random actions to explore new possibilities. As learning progresses and the agent gains more knowledge, the exploration rate decreases, and the agent becomes more likely to exploit known good actions.

f) Saving and Loading Model Parameters

During training, the DQN agent's neural network model improves its ability to estimate Q-values and make better decisions. To preserve this progress, it's important to save the model parameters periodically.

Saving the model parameters allows the agent to resume training from a specific point in time, avoiding the need to restart the entire learning process from scratch. Additionally, saved models can be used to evaluate the agent's performance on different tasks or environments.