Artificial Neural Network Concepts

If you're getting started with artificial neural networks or looking to expand your knowledge to new areas of the field, this page will give you a brief introduction to all the important concepts:

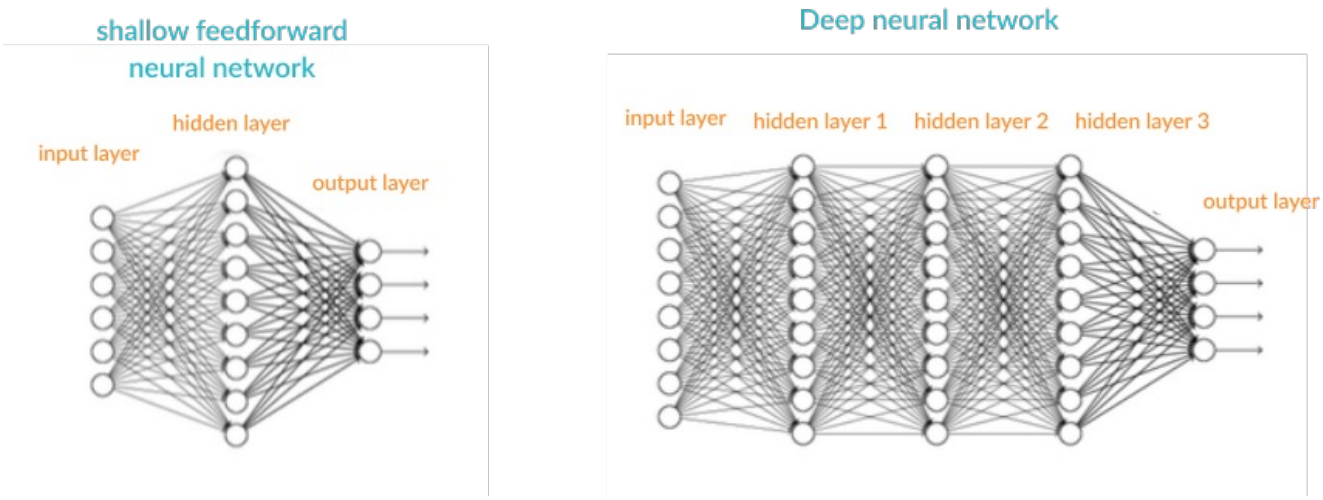
What are Artificial Neural Networks and Deep Neural Networks?

Artificial Neural Networks (ANN) is a supervised learning system built of a large number of simple elements, called neurons or perceptrons. Each neuron can make simple decisions, and feeds those decisions to other neurons, organized in interconnected layers. Together, the neural network can emulate almost any function, and answer practically any question, given enough training samples and computing power. A "shallow" neural network has only three layers of neurons:

- **An input layer** that accepts the independent variables or inputs of the model
- **One hidden layer**
- **An output layer** that generates predictions

A Deep Neural Network (DNN) has a similar structure, but it has two or more "hidden layers" of neurons that process inputs. Goodfellow, Bengio and Courville showed that while shallow neural networks are able to tackle complex problems, deep learning networks are more accurate, and improve in accuracy as more neuron layers are added. Additional layers are useful up to a limit of 9-10, after which their predictive power starts to decline. Today most neural network models and implementations use a deep network of between 3-10 neuron layers.

Shallow vs deep neural networks



Here is a glossary of basic terms you should be familiar with before learning the details of neural networks.

Inputs

Source data fed into the neural network, with the goal of making a decision or prediction about the data. Inputs to a neural network are typically a set of real values; each value is fed into one of the neurons in the input layer.

Training Set

A set of inputs for which the correct outputs are known, used to train the neural network.

Outputs

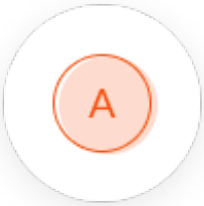
Neural networks generate their predictions in the form of a set of real values or boolean decisions. Each output value is generated by one of the neurons in the output layer.

Neuron/perceptron

The basic unit of the neural network. Accepts an input and generates a prediction.

Activation Function

Each neuron accepts part of the input and passes it through the activation function. Common activation functions are sigmoid, TanH and ReLu. Activation functions help generate output values within an acceptable range, and their non-linear form is crucial for training the network.



Weight Space

Each neuron is given a numeric weight. The weights, together with the activation function, define each neuron's output. Neural networks are trained by fine-tuning weights, to discover the optimal set of weights that generates the most accurate prediction.

Forward Pass

The forward pass takes the inputs, passes them through the network and allows each neuron to react to a fraction of the input. Neurons generate their outputs and pass them on to the next layer, until eventually the network generates an output.

Error Function

Defines how far the actual output of the current model is from the correct output. When training the model, the objective is to minimize the error function and bring output as close as possible to the correct value.

Backpropagation

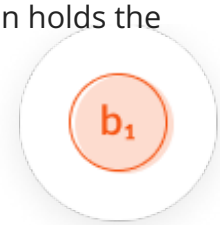
In order to discover the optimal weights for the neurons, we perform a backward pass, moving back from the network's prediction to the neurons that generated that prediction. This is called backpropagation.

Backpropagation tracks the derivatives of the activation functions in each successive neuron, to find weights that brings the loss function to a minimum, which will generate the best prediction. This is a mathematical process called *gradient descent*.

Bias and Variance

When training neural networks, like in other machine learning techniques, we try to balance between bias and variance. Bias measures how well the model fits the training set—able to correctly predict the known outputs of the training examples. Variance measures how well the model works with unknown inputs that were not available during training. Another meaning of bias is a "bias

neuron" which is used in every layer of the neural network. The bias neuron holds the number 1, and makes it possible to move the activation function up, down, left and right on the number graph.



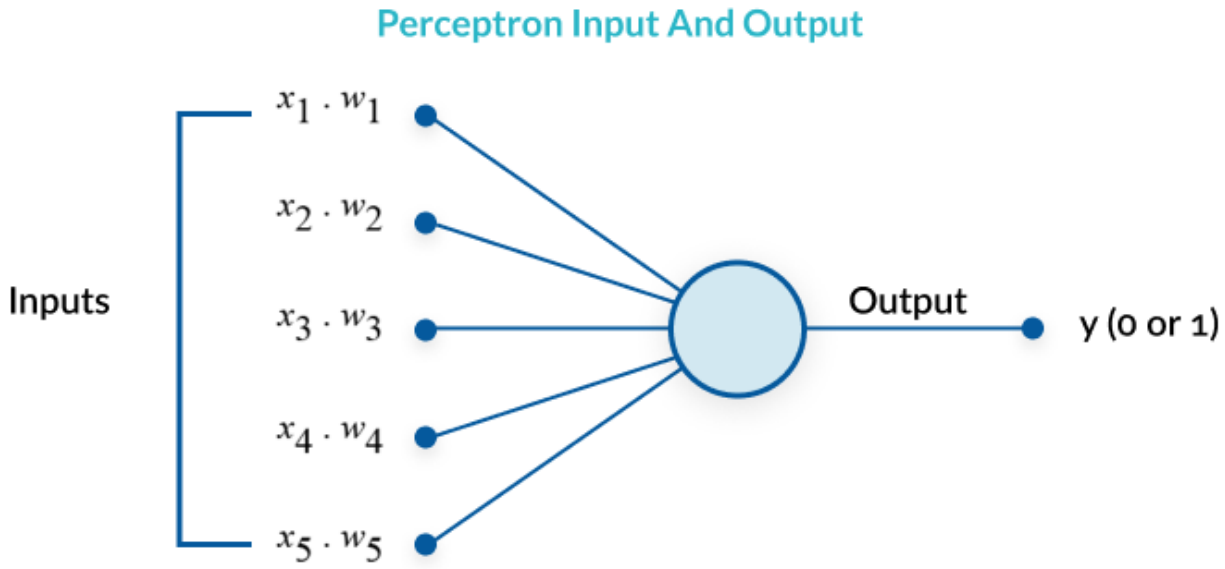
Hyperparameters

A hyperparameter is a setting that affects the structure or operation of the neural network. In real deep learning projects, tuning hyperparameters is the primary way to build a network that provides accurate predictions for a certain problem. Common hyperparameters include the number of hidden layers, the activation function, and how many times (epochs) training should be repeated.

Perceptron and Multilayer Perceptron—the Foundation of Neural Networks

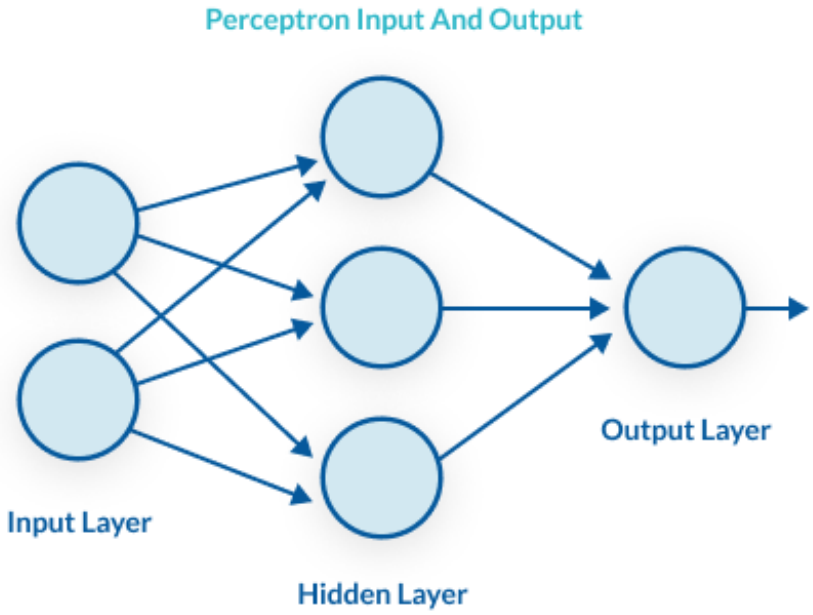
What is a Perceptron?

A perceptron is a binary classification algorithm modeled after the functioning of the human brain—it was intended to emulate the neuron. The perceptron, while it has a simple structure, has the ability to learn and solve very complex problems.

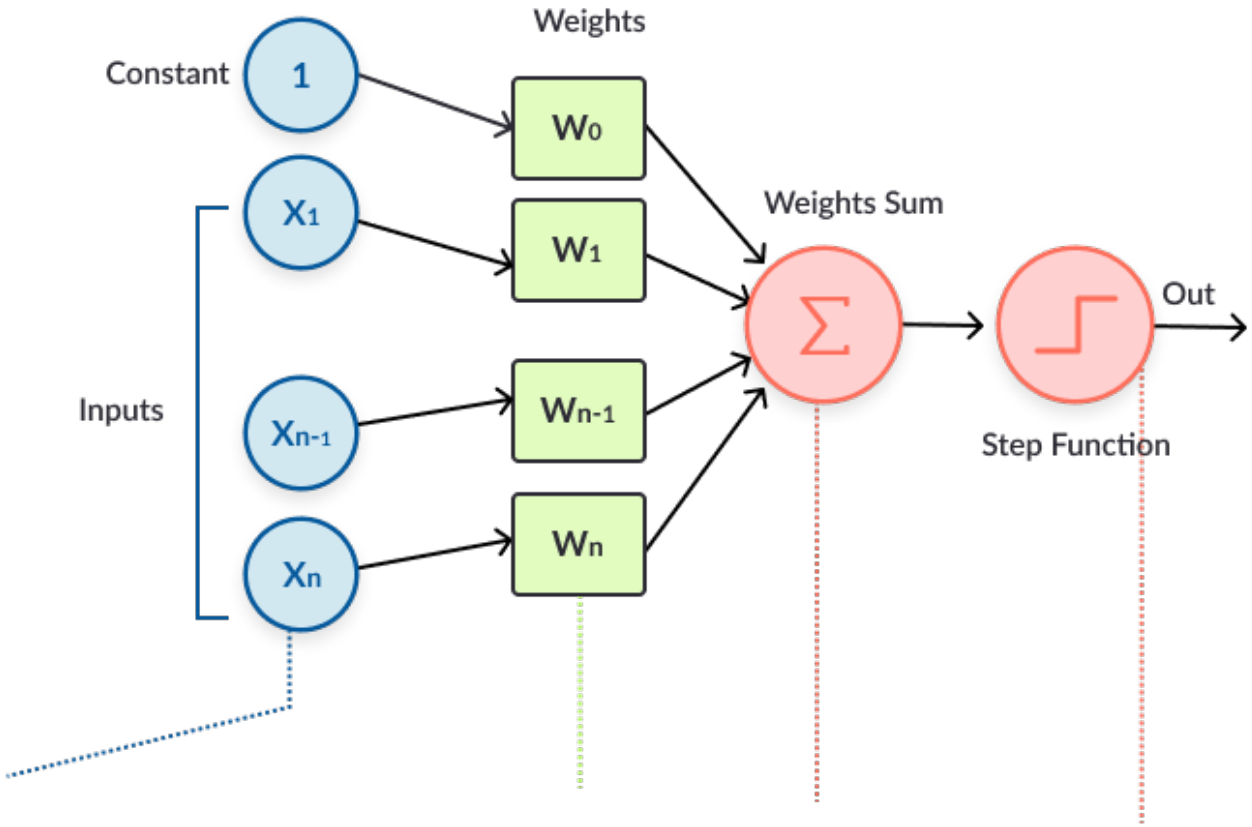


What is Multilayer Perceptron?

A multilayer perceptron (MLP) is a group of perceptrons, organized in multiple layers, that can accurately answer complex questions. Each perceptron in the first layer (on the left) sends signals to all the perceptrons in the second layer, and so on. An MLP contains an input layer, at least one hidden layer, and an output layer.



The Perceptron Learning Process



The perceptron learns as follows:

1. Takes the inputs which are fed into the perceptrons in the input layer, multiplies them by their weights, and computes the sum.
2. Adds the number one, multiplied by a "bias weight". This is a technical step that makes it possible to move the output function of each perceptron (the activation function) up, down, left and right on the number graph.
3. Feeds the sum through the activation function—in a simple perceptron system, the activation function is a step function.
4. The result of the step function is the output.

From Perceptron to Deep Neural Network

A multilayer perceptron is quite similar to a modern neural network. By adding a few ingredients, the perceptron architecture becomes a full-fledged deep learning system:

- **Activation functions and other hyperparameters** —a full neural network uses a variety of activation functions which output real values, not boolean values like in the classic perceptron. It is more flexible in terms of other details of the learning process, such as the number of training iterations (iterations and epochs), weight initialization schemes, regularization, and so on. All these can be tuned as hyperparameters.

- **Backpropagation**—a full neural network uses the backpropagation algorithm, to perform iterative backward passes which try to find the optimal values of perceptron weights, to generate the most accurate prediction.
- **Advanced architectures**—full neural networks can have a variety of architectures that can help solve specific problems. A few examples are Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), and Generative Adversarial Networks (GAN).

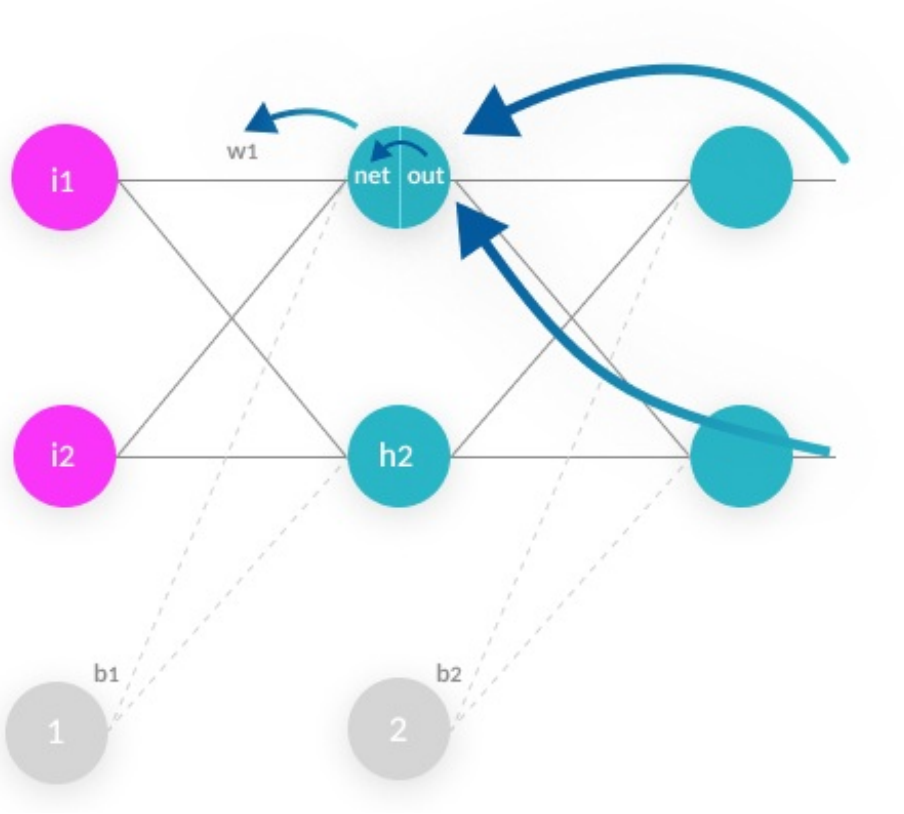
Go in-depth: *See our complete guide to perceptrons and multi-layer perceptrons*

Understanding Backpropagation in Neural Networks

What is Backpropagation and Why is it Important?

After a neural network is defined with initial weights, and a forward pass is performed to generate the initial prediction, there is an error function which defines how far away the model is from the true prediction. There are many possible algorithms that can minimize the error function—for example, one could do a brute force search to find the weights that generate the smallest error. However, for large neural networks, a training algorithm is needed that is very computationally efficient. Backpropagation is that algorithm—it can discover the optimal weights relatively quickly, even for a network with millions of weights.

How Backpropagation Works



1. **Forward pass**—weights are initialized and inputs from the training set are fed into the network. The forward pass is carried out and the model generates its initial prediction.
2. **Error function**—the error function is computed by checking how far away the prediction is from the known true value.
3. **Backpropagation with gradient descent**—the backpropagation algorithm calculates how much the output values are affected by each of the weights in the model. To do this, it calculates partial derivatives, going back from the error function to a specific neuron and its weight. This provides complete traceability from total errors, back to a specific weight which contributed to that error. The result of backpropagation is a set of weights that minimize the error function.
4. **Weight update**—weights can be updated after every sample in the training set, but this is usually not practical. Typically, a batch of samples is run in one big forward pass, and then backpropagation performed on the aggregate result. The *batch size* and number of batches used in training, called *iterations*, are important hyperparameters that are tuned to get the best results. Running the entire training set through the backpropagation process is called an *epoch*.

Backpropagation in the Real World

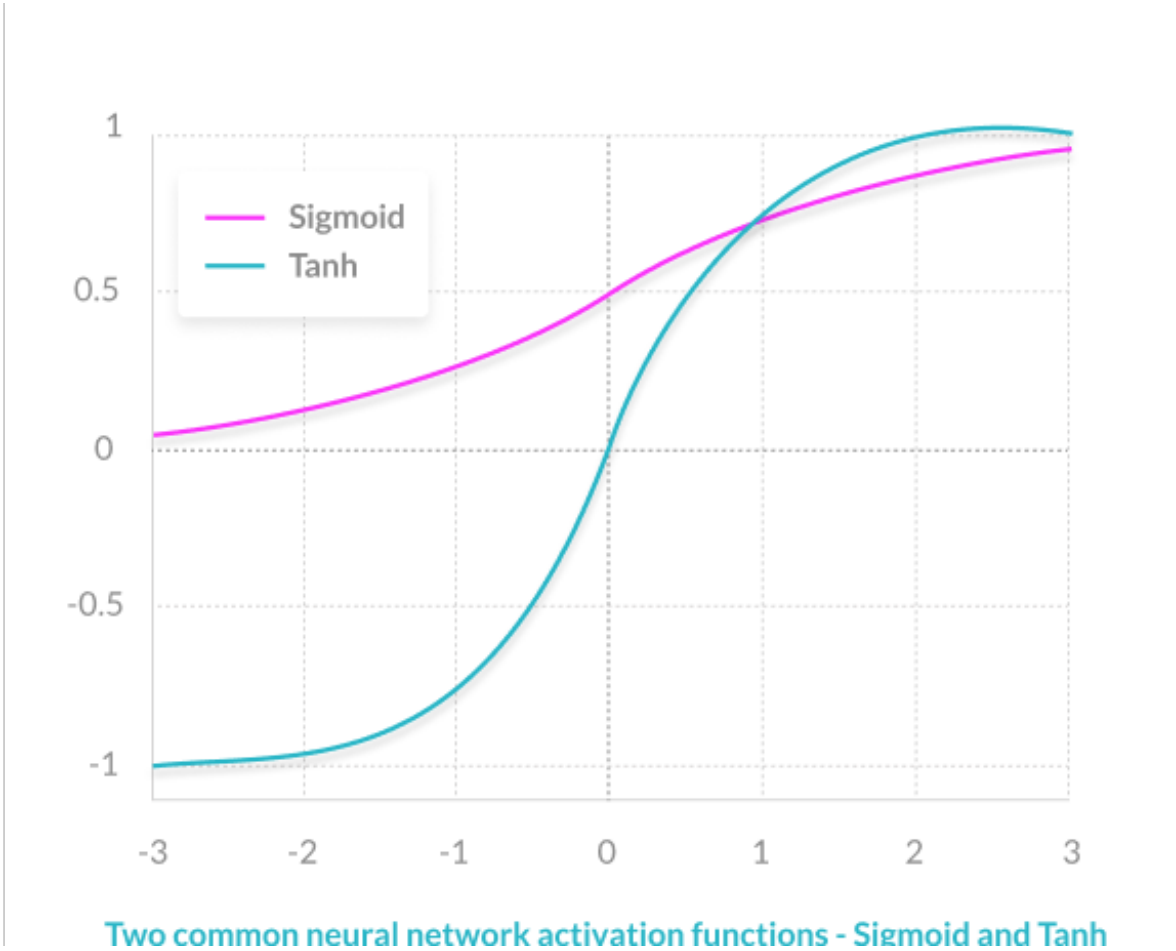
In the real world, you will probably not code an implementation of backpropagation, because others have already done this for you. You can work with deep learning frameworks like Tensorflow or Keras, which contain efficient implementations of backpropagation, which you can run with only a few lines of code. ***Go in-depth:*** *Learn more in our complete guide to backpropagation*

Understanding Neural Network Activation Functions

Activation functions are central to deep learning architectures. They determine the output of the model, its computational efficiency, and its ability to train and converge after multiple iterations of training.

What is a Neural Network Activation Function?

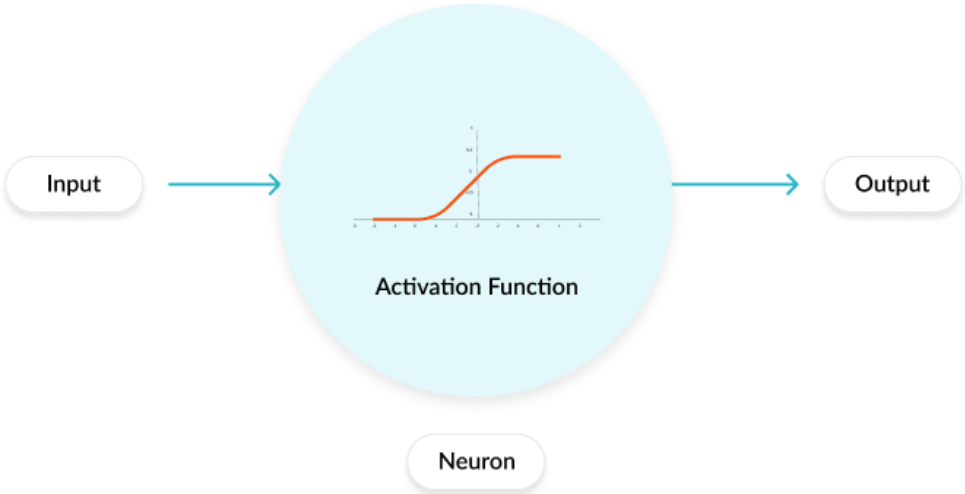
An activation function is a mathematical equation that determines the output of each element (perceptron or neuron) in the neural network. It takes in the input from each neuron and transforms it into an output, usually between one and zero or between -1 and one. Classic activation functions used in neural networks include the step function (which has a binary input), sigmoid and tanh. New activation functions, intended to improve computational efficiency, include ReLu and Swish.



Two common neural network activation functions - Sigmoid and Tanh

Role of the Activation Function

In a neural network, inputs, which are typically real values, are fed into the neurons in the network. Each neuron has a weight, and the inputs are multiplied by the weight and fed into the activation function.



Each neuron's output is the input of the neurons in the next layer of the network, and so the inputs cascade through multiple activation functions until eventually, the output layer generates a prediction. Neural networks rely on nonlinear activation functions—the derivative of the activation function helps the network learn through the backpropagation process (see backpropagation above).

7 Common Activation Functions

1. **The sigmoid function** has a smooth gradient and outputs values between zero and one. For very high or low values of the input parameters, the network can be very slow to reach a prediction, called the *vanishing gradient* problem.
2. **The TanH function** is zero-centered making it easier to model inputs that are strongly negative strongly positive or neutral.
3. **The ReLu function** is highly computationally efficient but is not able to process inputs that approach zero or negative.
4. **The Leaky ReLu** function has a small positive slope in its negative area, enabling it to process zero or negative values.
5. **The Parametric ReLu** function allows the negative slope to be learned, performing backpropagation to learn the most effective slope for zero and negative input values.
6. **Softmax** is a special activation function use for output neurons. It normalizes outputs for each class between 0 and 1, and returns the probability that the input belongs to a specific class.
7. **Swish** is a new activation function discovered by Google researchers. It performs better than ReLu with a similar level of computational efficiency.

Activation Functions in the Real World

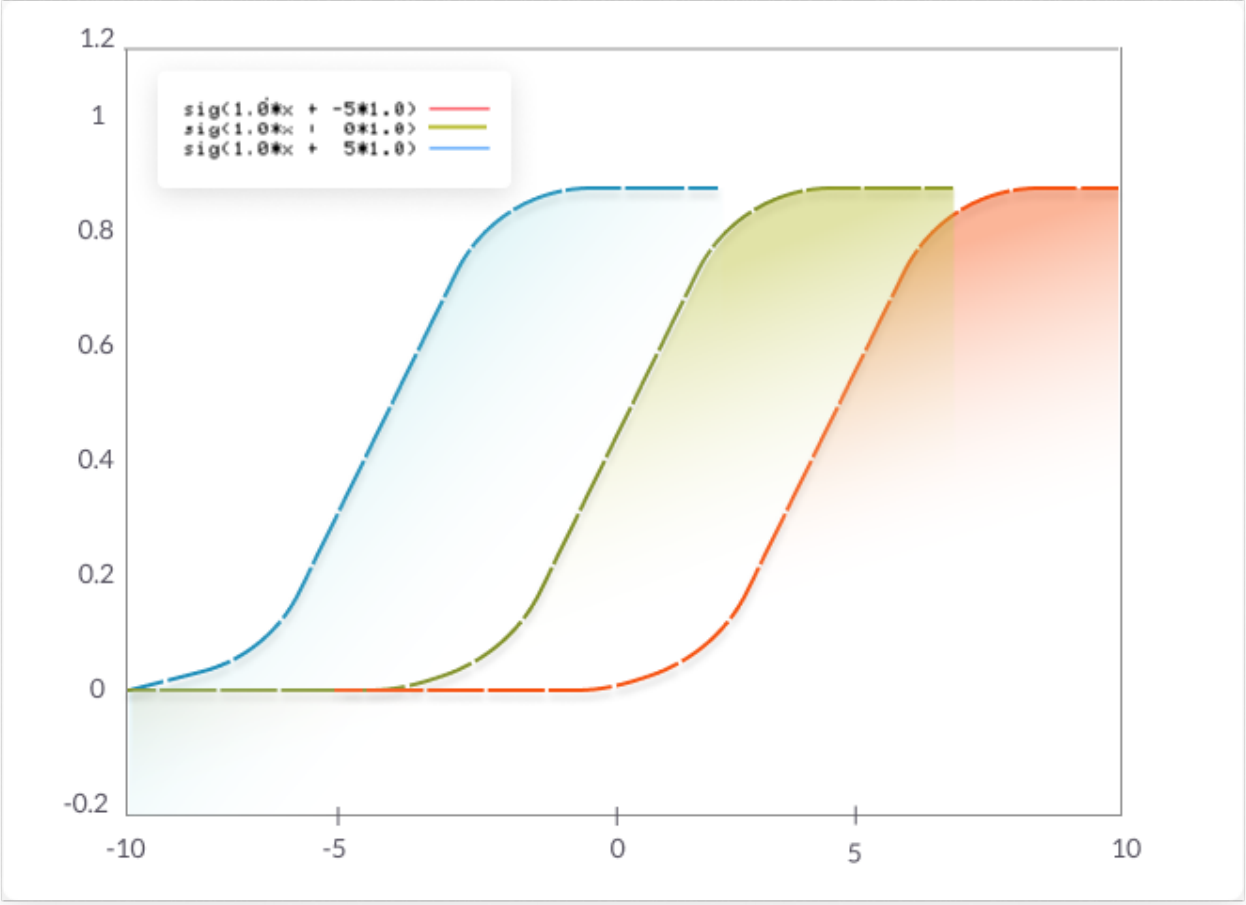
The selection of an activation function is critical to building and training in your network. Experimenting with different activation functions will allow you to achieve better results. In real-world neural network projects, the activation function is a hyperparameter. You can use the deep learning framework of your choice to change the activation function as you fine-tune your experiments. ***Go in-depth:*** *Learn more in our complete guide to neural network activation functions*

Neural Network Bias

In artificial neural networks, the word bias has two meanings:

- It can mean a *bias neuron*, which is part of the structure of the neural network
- It can mean *bias* as a statistical concept, which reflects how well the network is able to generate predictions based on the training samples you provide.

The bias neuron In each layer of the neural network, a bias neuron is added, which simply stores a value of 1. The bias neuron makes it possible to move the activation function left, right, up, or down on the number graph. Without a bias neuron, each neuron takes the input and multiplies it by its weight, without adding anything to the activation equation. This means, for example, it is not possible to input a value of zero and generate an output of two. In many cases it's necessary to move the entire activation function to the left or to the right, upwards or downwards, to generate the required output values; the bias neuron makes this possible.

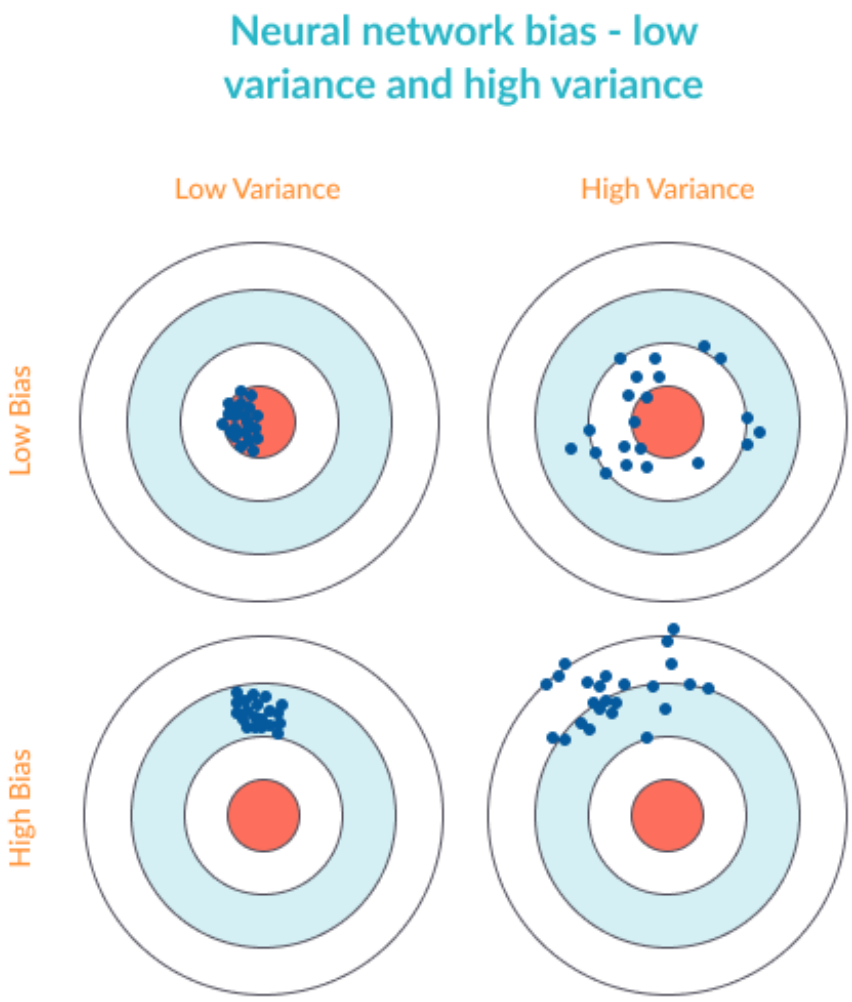


Definition of Bias vs. Variance in Neural Networks

To understand bias vs. variance, we first need to introduce the concept of a training set and validation set:

- A **training set** is a group of examples which is fed to the neural network during training.
- A **validation set** is a group of unseen examples which you use to test your neural network to see how it performs.
- An **error function** calculates the error, for either the training or validation sets. The eror reflects how far away the network's actual predictions were compared to the known correct outputs.

Bias reflects how well the model fits the training set. A high bias means the neural network is not able to generate correct predictions even for the examples it trained on. **Variance** reflects how well the model fits unseen examples in the validation set. A high variance means the neural network is not able to correctly predict for new examples it hasn't seen before.



Overfitting and Underfitting in Neural Networks

Overfitting happens when the neural network is good at learning its training set, but is not able to generalize its predictions to additional, unseen examples. This is characterized by low bias and high variance. **Underfitting** happens when the neural network is not able to accurately predict for the training set, not to mention for the validation set. This is characterized by high bias and high variance.

Methods to Avoid Overfitting

Here are a few common methods to avoid overfitting in neural networks:

- **Retraining neural networks**—running the same model on the same training set but with different initial weights, and selecting the network with the best performance.
- **Multiple neural networks**—training several neural network models in parallel, with the same structure but different weights, and averaging their outputs.

- **Early stopping**—training the network, monitoring the error on the validation set after each iteration, and stopping training when the network starts to overfit the data.
- **Regularization**—adding a term to the error function equation, intended to decrease the weights and biases, smooth outputs and make the network less likely to overfit.
- **Tuning performance ratio**—similar to regularization, but using a parameter that defines by how much the network should be regularized.

Methods to Avoid Underfitting

Here are a few common methods to avoid underfitting in a neural network:

- **Adding neuron layers or inputs**—adding neuron layers, or increasing the number of inputs and neurons in each layer, can generate more complex predictions and improve the fit of the model.
- **Adding more training samples or improving quality**—the more training samples you feed into the network, and the better they represent the variance in the real population, the better the network will perform.
- **Dropout**—randomly "kill" a certain percentage of neurons in every training iteration. This ensures some information learned is randomly removed, reducing the risk of overfitting.
- **Decreasing regularization parameter**—regularization can be overdone. By using a regularization performance parameter, you can learn the optimal degree of regularization, which can help the model better fit the data.

Go in-depth: *Learn more in our complete guide to neural network bias*

Neural Network Hyperparameters

Hyperparameters determine how the neural network is structured, how it trains, and how its different elements function. Optimizing hyperparameters is an art: there are several ways, ranging from manual trial and error to sophisticated algorithmic methods.

The Difference Between Model Parameter and Hyperparameter

- **A model parameter** is internal to learn your network and is used to make predictions in a production deep learning model. The objective of training is to learn the values of the model parameters.
- **A hyperparameter** is an external parameter set by the operator of the neural network. For example, the number of iterations of training, the number of hidden layers, or the activation function. Different values of hyperparameters can have a major impact on the performance of the network.

List of Common Hyperparameters

Hyperparameters related to neural network structure	Hyperparameters related to the training algorithm
• Number of hidden layers • Dropout • Activation function • Weights initialization	• Learning rate • Epoch, iterations and batch size • Optimizer algorithm • Momentum

4 Methods of Hyperparameter Tuning

In a neural network experiment, you will typically try many possible values of hyperparameters and see what works best. In order to evaluate the success of different values, retrain the network, using each set of hyperparameters, and test it against your validation set. If your training set is small, you can use **cross-validation**—dividing the training set into multiple groups, training the model on each of the groups then validating it on the other groups. Following are common methods used to tune hyperparameters:

1. **Manual hyperparameter tuning**—an experienced operator can guess parameter values that will achieve very high accuracy. This requires trial and error.
2. **Grid search**—this involves systematically testing multiple values of each hyperparameter and retraining the model for each combination.
3. **Random search**—a research study by Bergstra and Bengio showed that using random hyperparameter values is actually more effective than manual search or grid search.
4. **Bayesian optimization—**a method proposed by Shahriari, et al, which trains the model with different hyperparameter values over and over again, and tries to observe the shape of the function generated by different parameter values. It then extends this function to predict the best possible values. This method provides higher accuracy than random search.

Hyperparameter Optimization in the Real World

In a real neural network project, you can either manually optimize hyperparameter values; use optimization techniques in the deep learning framework of your choice, or use one of several third-party hyperparameter optimization tools. **If you use Keras**, you can use these libraries for hyperparameter optimization: Hyperopt, Kopt and Talos **If you use TensorFlow**, you can use GPflowOpt for bayesian optimization, and commercial solutions like Google's Cloud Machine Learning Engine which provide multiple optimization options. **For third-party optimization tools**, see this post by Mikko Kotila. ***Go in-depth:*** *Learn more in our complete guide to hyperparameters*

Classification with Neural Networks

What is classification in machine and deep learning?

There are numerous, highly effective classification algorithms; neural networks are just one of them. The unique strength of a neural network is its ability to dynamically create complex prediction functions, and solve classification problems in a way that emulates human thinking. For certain classification problems, neural networks can provide improved performance compared to other algorithms. However, because neural networks are more computationally intensive and more complex to set up, they may be overkill in many cases.

Types of classification algorithms

To understand classification with neural networks let's cover some other common classification algorithms. Some algorithms are binary, providing a yes/no decision, while others are multi-class, letting you classify an input into several categories.

- **Logistic regression** (binary)—analyzes a set of data points and finds the best fitting model to describe them. Easy to implement and very effective for input variables that are well known, and closely correlated with the outcome.
- **Decision tree** (multiclass)—classifies using a tree structure with if-then rules, running the input through a series of decisions until it reaches a termination condition. Able to model complex decision processes and is highly intuitive, but can easily overfit the data.
- **Random forest** (multiclass)—an ensemble of decision trees, with automatic selection of the best performing tree. Provides the strength of the decision tree algorithm without the problem of overfitting.
- **Naive Bayes classifier** (multiclass)—a probability-based classifier. Calculates the likelihood that each data point exists in each of the target categories. Simple to implement and accurate for a large set of problems, but sensitive to the set of categories selected.
- **k-Nearest neighbor** (multiclass)—classifies each data point by analyzing its nearest neighbors among the training examples. Simple to implement and understand, effective for many problems, especially those with low dimensionality. Provides lower accuracy compared to supervised algorithms, and is computationally intensive.

Neural Network Classification Compared To Other Classifier Algorithms

Neural networks classify by passing the input values through a series of neuron layers, which perform complex transformations on the data. **Strengths:** Neural networks are very effective for high dimensionality problems, or with complex relations between variables. For

example, neural networks can be used to classify and label images, audio, and video, perform sentiment analysis on text, and classify security incidents into risk categories. **Weaknesses:** Neural networks are theoretically complex, difficult to implement, requiring careful fine-tuning, and computationally intensive. Unless you're a deep learning expert, you will usually derive more value from another classification algorithm if it can provide similar performance.

Neural Networks and Other Machine Learning Classifiers in the Real World

In a real-world machine learning project, you will probably experiment with several classification algorithms to see which provides the best result. **If you restrict yourself to regular classifiers** (not neural networks), you can use open source libraries like scikit-learn, which provides ready-made implementations of popular algorithms and is easy to get started with. **If you want to try neural network classification**, you will need to use deep learning frameworks like TensorFlow, Keras, and PyTorch. These frameworks are very powerful, supporting both neural networks and traditional classifiers like naive bayes, but have a steeper learning curve. ***Go in-depth:*** *Learn more in our complete guide to classification with neural networks*

Using Neural Networks for Regression

What is Regression Analysis?

For decades, regression models have proven useful in modeling problems and providing predictions. This is the classic linear regression function:

$$\gamma = \beta_1 + \beta_2 X_2 + \beta_3 X_3 + \cdots + \beta_k X_k + \varepsilon$$

In a regression model, the inputs are called **independent values** (X1..K in the equation above). The output is called the **dependent value** (y in the equation above). There are weights called **coefficients**, which determine how much each input value contributes to the result, or how important it is (β1..K in the equation above). Neural networks are able to model complex problems, using a learning process that emulates the human brain. Can you use a neural network to run a regression? The short answer is yes – neural networks can generate a model that approximates any regression function. Moreover, most regression models do not fit the data perfectly, and neural networks can generate a more complex model that will provide higher accuracy.

Types of Regression Analysis

- **Linear regression**—suitable for dependent values which can be fitted with a straight

line (linear function).

- **Polynomial regression**—suitable for dependent variables which can be fitted by a curve or series of curves.
- **Logistic regression**—suitable for dependent variables which are binary, and therefore not normally distributed.
- **Stepwise regression—**an automated technique that can deal with high dimensionality of independent variables.
- **Ridge regression**—a regression technique that helps with multicollinearity, independent variables that are highly correlated. It adds a bias to the regression estimates, penalizing coefficients using a shrinkage parameter.
- **Lasso regression**—like Ridge regression, shrinks coefficients to solve multicollinearity, however, it also shrinks the absolute values, meaning some of the coefficients can become zero. This performs "feature selection", removing some variables from the equation.
- **ElasticNet regression**—combines Ridge and Lasso regression, and is trained with L1 and L2 regularization, trading off between the two techniques.

Regression in Neural Networks

Neural networks are a far more complex mathematical structure than regression models, but they are reducible to regression equations. Essentially, any regression equation can be modeled by a neural network. For example, this very simple neural network, which takes them several inputs, multiplies them by weights, and passes them through a step function, is equivalent to a logistic regression. A slightly more complex neural network can be constructed to model a multi-class regression classification, using the Softmax activation function to generate probabilities for each class, which can be normalized to sum up to 1.

Should Neural Networks be Used to Run Regression Models?

Neural networks can be used to create regression models. But is it worthwhile to use them for this purpose? The answer depends on your intuition regarding the effectiveness of the regression function:

- For some datasets and problems, regression functions can provide very accurate answers. In these cases, a neural network is probably not needed.
- If the data is complex and the regression function cannot model it accurately, a neural network can create a more complex mathematical structure that will accurately represent the data. In these more complex cases, neural networks can have much more predictive power and can perform much better than a regression function.

Regression with Neural Networks in Real Life

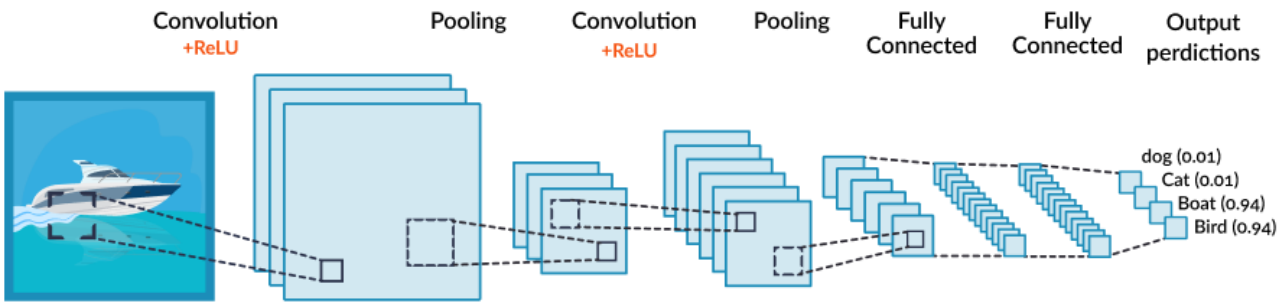
To run a traditional regression function, you would typically use R or another mathematics or statistics library. In order to run a neural network equivalent to a regression model, you will need to use deep learning frameworks, such as TensorFlow, Keras or PyTorch, which are more difficult to master. While neural networks have their overhead and are more theoretically complex, they provide prediction power uncomparable to the most sophisticated regression models. Regression equations are limited and cannot perfectly fit all expected data sets, and the more complex your scenario, the more you will benefit from entering the world of deep learning. ***Go in-depth:*** *Learn more in our complete guide to using neural networks for regression*

Neural Network Architecture

We covered the traditional or "plain vanilla" Artificial Neural Network architecture in previous sections: Multilayer Perceptrons and Understanding Backpropagation. On top of this basic structure, researchers have proposed several advanced architectures. Below we cover several architectures which are widely deployed and help provide answers to questions that are difficult to solve with a traditional neural network structure.

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) have proven very effective at tasks involving data that is closely knitted together, primarily in the field of computer vision. A CNN uses a three-dimensional structure, with three sets of neurons analyzing the three layers of a color image —red, green and blue. It analyzes an image one area at a time to identify important features. **CNN Architecture** The "fully connected" neural network structure, in which all neurons in one layer communicate with all the neurons in the next layer, is inefficient when it comes to analyzing large images. A CNN uses a three-dimensional structure in which neurons in one layer do not connect to all the neurons in the next layer, instead, each set of neurons analyzes a small region or "feature" of the image. The final output of this structure is a single vector of probability scores. A CNN first performs a *convolution*, which involves "scanning" the image, analyzing a small part of it each time, and creating a *feature map* with probabilities that each feature belongs to the required class (in a simple classification example). The second step is *pooling*, which reduces the dimensionality of each feature while maintaining its most important information.



As illustrated above, a CNN can perform several rounds of convolution then pooling. Finally, when the features are at the right level of granularity, it creates a fully-connected neural network that analyzes the final probabilities and decides which class the image belongs to. The final step can also be used for more complex tasks, such as generating a caption for the image. **What can a CNN do? A few example applications:**

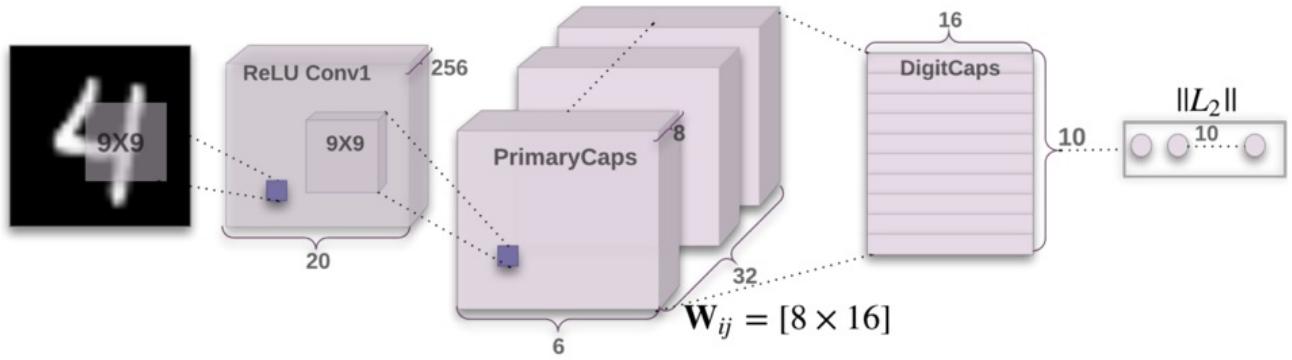
- **Face recognition**
- **Identifying and classifying everyday objects in images**
- **Powering vision in robots and autonomous vehicles**
- **Recognizing scenes and suggest relevant captions**
- **Semantic parsing, sentence classification, and prediction**
- **Search query retrieval**

To learn more about implementing CNNs, see Convolutional Neural Network: How to Build One in Keras & PyTorch

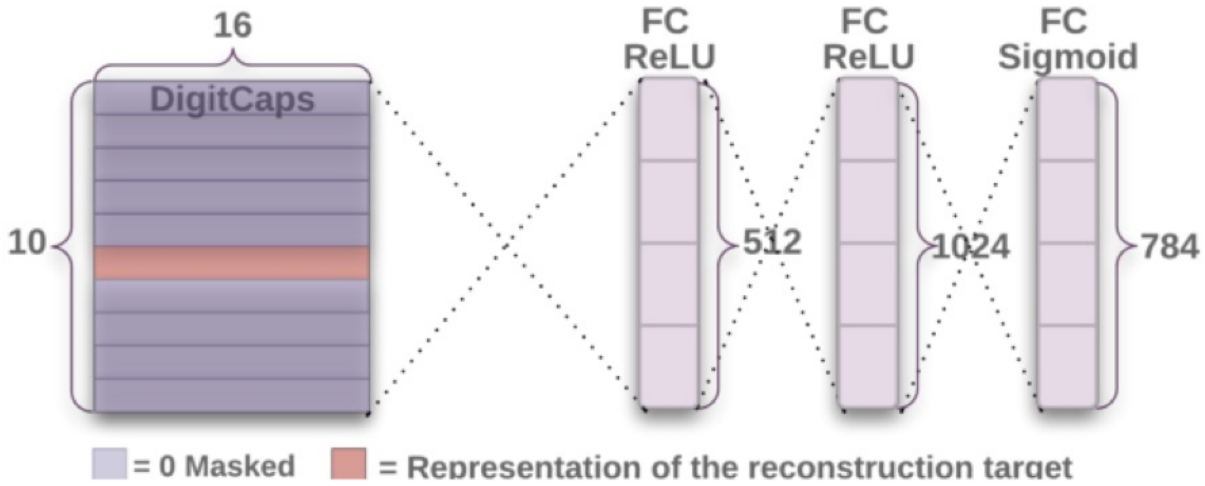
CAPSNet

CAPSNet is a new architecture proposed in 2017, which aims to solve a problem of Convolutional Neural Networks (CNN). CNNs are good at classifying images, but they fail when images are rotated or tilted, or when an image has the features of the desired object, but not in the correct order or position, for example, a face with the nose and mouth switched around. The reason a CNN has trouble classifying these types of images is that it performs multiple phases of convolution and pooling. The pooling step summates and reduces the information in each feature discovered in the image. In this process, it loses important information such as the position of the feature and its relation to other features. For “ordinary” images this works well, but when images are not presented as expected, e.g. tilted sideways, the network will not classify the image into its correct class. **CAPSNet Architecture** CAPSNet is based on the concept of neural “capsules”. It starts with the convolution step just like a regular CNN. But instead of the pooling step, when the network discovers features in the image, it reshapes them into vectors, “squashes” them using a special activation function, and feeds each feature into a *capsule*. This is a specialized neural structure that deals only with this feature. Each capsule in the first layer begins processing and then feeds its result to one or more levels of secondary capsules, nested within the first

capsule. This is called *routing by agreement*. The primary capsule detects the learned features (e.g. left ear, right ear, nose), preserving contextual information like position and relation to other elements.



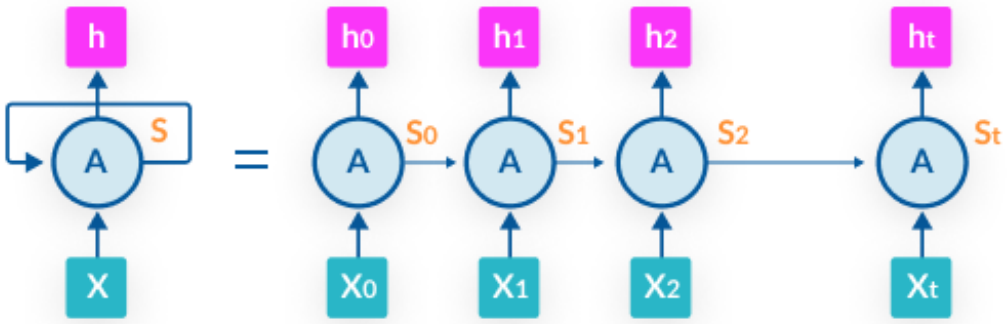
Encoder Architecture. Image Source: Dynamic Routing Between Capsules Paper The second part of the CAPSNet structure, called the *decoder*, uses the result of each capsule to recreate the image, based on the learned features. This final image is run through three fully-connected layers to perform the final classification.



Decoder Architecture. Image Source: Dynamic Routing Between Capsules Paper

Recurrent Neural Networks (RNN)

A Recurrent Neural Network (RNN) helps neural networks deal with input data that is sequential in nature. For example, written text, video, audio, or multiple events that occur one after the other, as in networking or security analytics. An RNN network accepts a series of inputs, remembers the previous inputs, and with each new input, adds a new layer of understanding. **RNN Architecture**



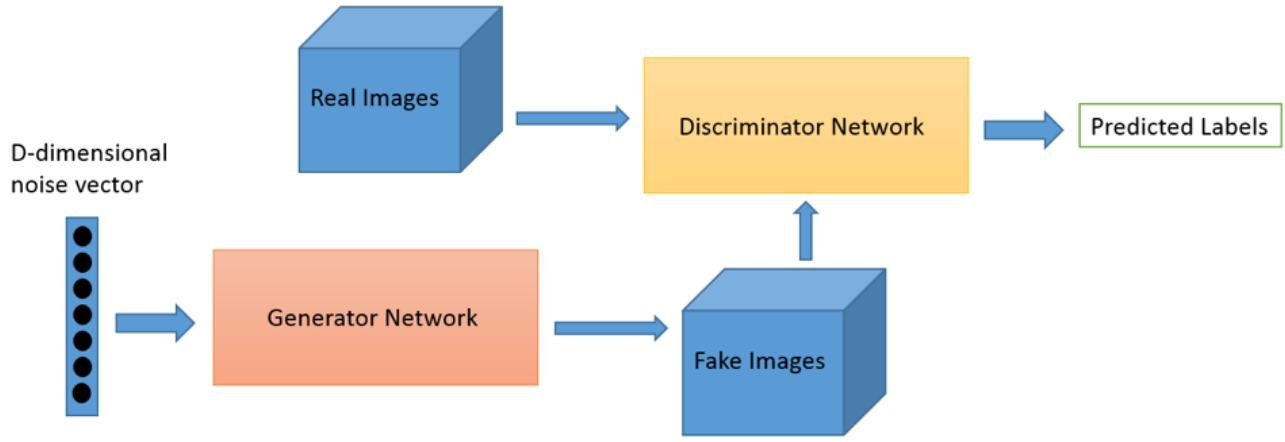
Unrolled Recurrent Neural Network

An RNN looks at a series of inputs over time, X0, X1, X2, until Xt. For example, this could be a sequence of frames in a video or words in a sentence. The neural network has one layer of neurons for each input. When the RNN network learns, it performs Backpropagation Through Time (BPTT), a multi-layered form of backpropagation. BPTT uses the chain rule to go back from the latest time step (Xt), progressively to each previous step, each time using gradient descent to discover the best weights for each neuron, and also learn the optimal weights that govern the transfer of information between one time step to the next. **What can an RNN Do?**

- **Language modeling and text generation**—models work by predicting the most suitable next character, next word, or next phrase in a string of text.
- **Machine translation**—text in the source language is input in batch, and the model attempts to generate matching text in the target language.
- **Speech recognition**—the input is acoustic signals parsed from an audio recording; the model outputs the most likely language phonetic element for each part of the recording.
- **Generating image captions**—the input is an image, and the model identifies features in the image and generates text that describes it.
- **Time series anomaly detection**—the input is a sequential data series, such as a series of events in a potential cybersecurity incident. The model predicts if the data is an anomaly compared to normal behavior.
- **Video tagging**—the input is a series of video frames, and the model generates a textual description of each frame of the video.

Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GAN) allow neural networks to generate photos, paintings and other artifacts that closely resemble real ones created by humans. It uses two neural networks, one of which generates sample images, and another which learns how to discriminate auto-generated images from real images. The closed feedback loop between the two networks makes them better and better at generating fake artifacts that resemble real ones. **GAN Architecture** GAN mimics images by pitting two neural networks against each other, one a convolutional neural network, the "generator", and the other a deconvolutional neural network, the "discriminator." The generator starts from random noise and creates new images, passing them to the discriminator, in the hope they will be deemed authentic (even though they are fake). The discriminator aims to identify images coming from the generator as fake, distinguishing them from real images. In the beginning, this is easy, but it becomes harder and harder. The discriminator learns based on the ground truth of the image samples which it knows. The generator learns from the feedback of the discriminator—if the discriminator "catches" a fake image, the generator tries harder to emulate the source images.



What can a GAN do? GANs can automatically generate or enhance:

- Photos
- Paintings
- Prose
- Speech
- Video

Examples of practical implications are generating art very similar to art by famous painters, visualizing how a person might look when they are old, visualizing industrial or interior design models, constructing 3D models from images, and improving quality of low-resolution images.

Artificial Neural Networks in the Real World

In this page, we covered many fundamental concepts of neural networks, including perceptron learning, backpropagation, activation functions, bias and variance, hyperparameters, and basic applications like classification and regression. We also covered several common advanced architectures, built on the basic neural network structure to solve new types of problems. ***This is just the introduction to a series of in-depth guides about neural network concepts***

> *To learn about these concepts in more depth, please see links to all our guide pages in the section below.*

Once you understand the mechanics of neural networks, and start working on real-life projects, you will use deep learning frameworks such as Keras, TensorFlow, and PyTorch to create, train and evaluate neural network models. These frameworks do not require an in-depth understanding of the mathematical structure of the models; they will allow you to create very complex neural network structures in only a few lines of code. Your focus will be on collecting high-quality training examples, selecting the best neural network architecture for your problem, and tuning hyperparameters to achieve the best results. When you start running models at scale using deep learning frameworks, you will run into a few challenges:

Tracking experiment progress, metrics, hyperparameters and code, as you perform trial and error to find the best neural structure for your problem.

Running experiments across multiple machines—neural networks are computationally intensive, and to work at large scale you will need to deploy them across several machines. Setting up and configuring these machines, and distributing work between them, is a major effort.

Manage training data—deep learning projects, especially those involving image or video analysis, can have very large training sets, from Gigabytes to Petabytes in size. You will find it complex to manage this training data, copying it to multiple machines, then erasing and replacing with fresh data.

Backpropagation in Neural Networks: Process, Example & Code

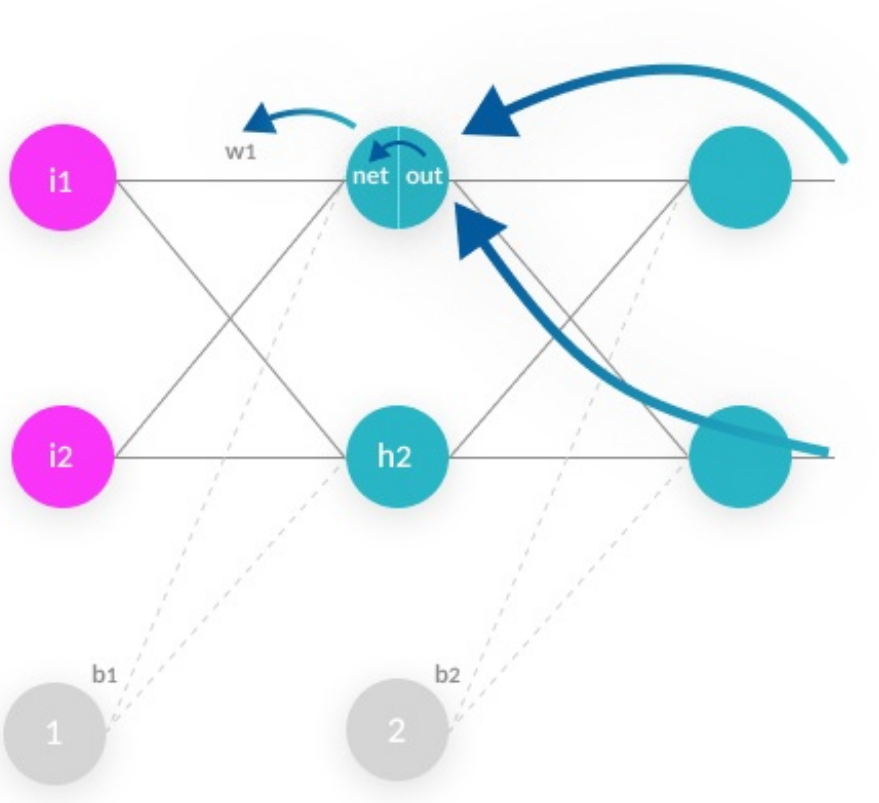
missinglink.ai/guides/neural-network-concepts/backpropagation-neural-networks-process-examples-code-minus-math

Backpropagation is a basic concept in modern neural network training. In real-world projects, you will not perform backpropagation yourself, as it is computed out of the box by deep learning frameworks and libraries. But it's very important to get an idea and basic intuitions about what is happening under the hood. This article is part of MissingLink's Neural Network Guide, which focuses on practical explanations of concepts and processes, skipping the theoretical or mathematical background.

In this article you'll learn:

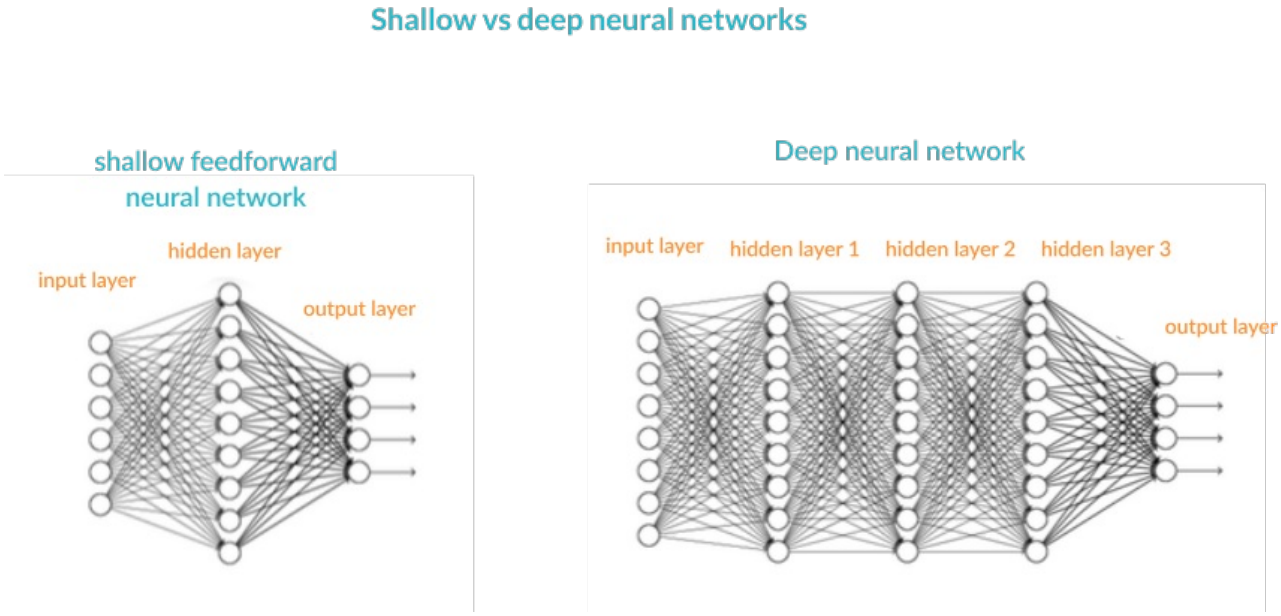
What is Backpropagation?

Backpropagation is an algorithm commonly used to train neural networks. When the neural network is initialized, weights are set for its individual elements, called neurons. Inputs are loaded, they are passed through the network of neurons, and the network provides an output for each one, given the initial weights. Backpropagation helps to adjust the weights of the neurons so that the result comes closer and closer to the known true result.



What are Artificial Neural Networks and Deep Neural Networks?

Artificial Neural Networks (ANN) are a mathematical construct that ties together a large number of simple elements, called neurons, each of which can make simple mathematical decisions. Together, the neurons can tackle complex problems and questions, and provide surprisingly accurate answers. A shallow neural network has three layers of neurons that process inputs and generate outputs. A Deep Neural Network (DNN) has two or more "hidden layers" of neurons that process inputs. According to Goodfellow, Bengio and Courville, and other experts, while shallow neural networks can tackle equally complex problems, deep learning networks are more accurate and improve in accuracy as more neuron layers are added.



ANN and DNN Concepts Relevant to Backpropagation

Here are several neural network concepts that are important to know before learning about backpropagation:

Inputs

Source data fed into the neural network, with the goal of making a decision or prediction about the data. The data is broken down into binary signals, to allow it to be processed by single neurons—for example an image is input as individual pixels.

Training Set

A set of outputs for which the correct outputs are known, which can be used to train the neural networks.

Outputs

The output of the neural network can be a real value between 0 and 1, a boolean, or a discrete value (for example, a category ID).

Activation Function

Each neuron accepts part of the input and passes it through the activation function. Commonly used functions are the sigmoid function, tanh and ReLu. Modern activation functions normalize the output to a given range, to ensure the model has stable convergence.

Weight Space

Each neuron is given a numeric weight. The weights, applied to the activation function, determine each neuron's output. In training of a deep learning model, the objective is to discover the weights that can generate the most accurate output.

Initialization

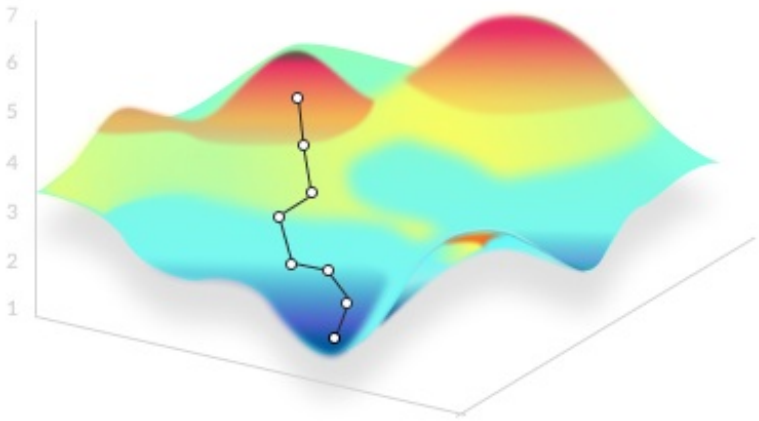
Setting the weights at the beginning, before the model is trained. A typical strategy in neural networks is to initialize the weights randomly, and then start optimizing from there. Xavier optimization is another approach which makes sure weights are "just right" to ensure enough signal passes through all layers of the network.

Forward Pass

The forward pass tries out the model by taking the inputs, passing them through the network and allowing each neuron to react to a fraction of the input, and eventually generating an output.

Gradient Descent

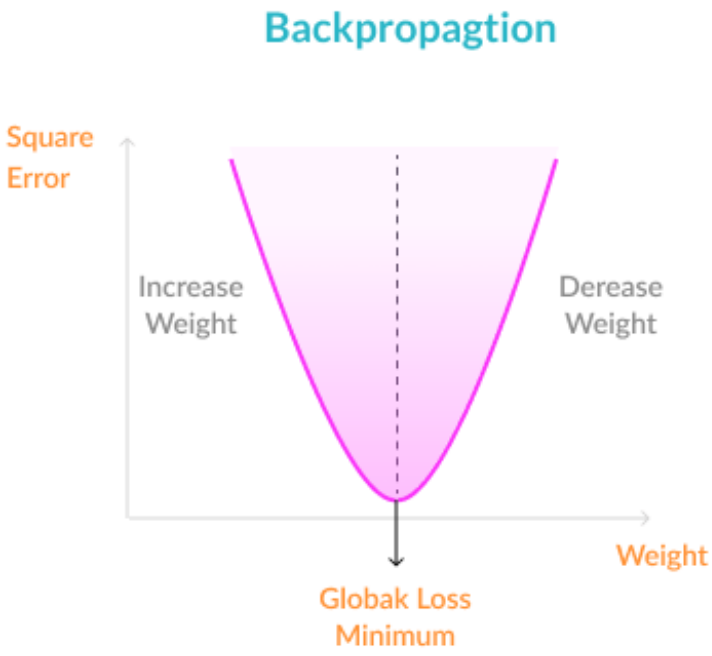
A mathematical technique that modifies the parameters of a function to descend from a high value of a function to a low value, by looking at the derivatives of the function with respect to each of its parameters, and seeing which step, via which parameter, is the next best step to minimize the function. Applying gradient descent to the error function helps find weights that achieve lower and lower error values, making the model gradually more accurate.



6 Stages of Neural Network Learning

Generally speaking, neural network or deep learning model training occurs in six stages:

1. **Initialization**—initial weights are applied to all the neurons.
2. **Forward propagation**—the inputs from a training set are passed through the neural network and an output is computed.
3. **Error function**—because we are working with a training set, the correct output is known. An error function is defined, which captures the delta between the correct output and the actual output of the model, given the current model weights (in other words, "how far off" is the model from the correct result).
4. **Backpropagation**—the objective of backpropagation is to change the weights for the neurons, in order to bring the error function to a minimum. Learn more below.



5. **Weight update—**weights are changed to the optimal values according to the results of the backpropagation algorithm.
6. **Iterate until convergence**—because the weights are updated a small delta step at a time, several iterations are required in order for the network to learn. After each iteration, the gradient descent force updates the weights towards less and less global loss function. The amount of iterations needed to converge depends on the learning rate, the network meta-parameters, and the optimization method used.

At the end of this process, the model is ready to make predictions for unknown input data. New data can be fed to the model, a forward pass is performed, and the model generates its prediction.

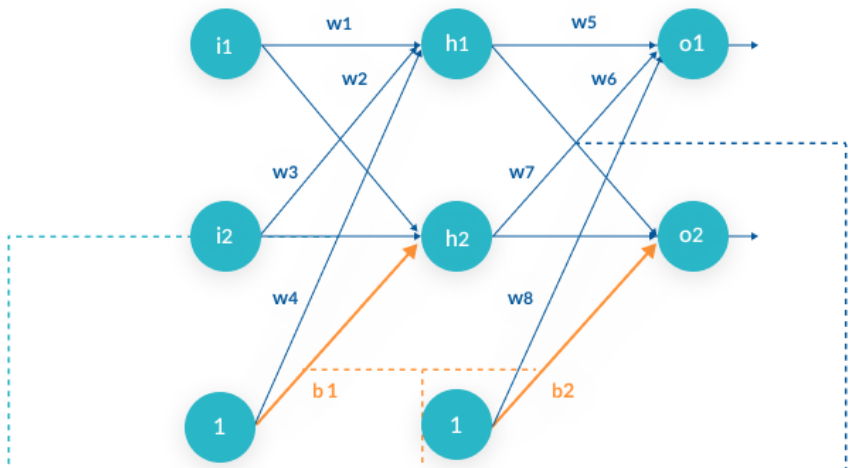
Why Do We Need Backpropagation in Neural Networks?

In the six stages of learning we presented above, step #4 can be done by any optimization function that can reduce the size of the error in the model. For example, you could do a brute force search to try to find the weight values that bring the error function to a minimum. Brute force or other inefficient methods could work for a small example model. But in a realistic deep learning model which could have as its output, for example, 600X400 pixels of an image, with 3-8 hidden layers of neurons processing those pixels, you can easily reach a model with millions of weights. This is why a more efficient optimization function is needed. Backpropagation is simply an algorithm which performs a highly efficient search for the optimal weight values, using the gradient descent technique. It allows you to bring the error functions to a minimum with low computational resources, even in large, realistic models.

How Backpropagation Works

We'll explain the backpropagation process in the abstract, with very simple math. To understand the mathematics behind backpropagation, refer to Sachin Joglekar's excellent post.

How Backpropagation Works



This model has eight weights

Remember—each neuron is a very simple component which does nothing but execute the activation function. There are several commonly used activation functions; for example, this is the sigmoid function:

$$f(x) = 1 / 1 + \exp(-x)$$

The model also has two biases

Biases in neural networks are extra neurons added to each layer, which store the value of 1. This allows you to "move" or translate the activation function so it doesn't cross the origin, by adding a constant number.

Without a bias neuron, each neuron can only take the input and multiply it by a weight. So, for example, it would not be possible to input a value of 0 and output 2. In many cases, it is necessary to move the entire activation function to the left or right to generate the required output values - this is made possible by the bias.

Go in-depth: See our guide on neural network bias

The model also has two biases

- Two weights are applied when input i1 is fed to the two hidden neurons, h1 and h2. These are called w1 and w2.
- Two weights are applied when input i2 is fed to the two hidden neurons—w3 and w4.
- Two more weights are applied when the result of the first hidden neuron, h1, is fed to the two output neurons, o1 and o2—w5 and w6.
- The last two weights are applied when the second hidden neuron, h2, feeds its result to the output neurons—w7 and w8.

What the neurons do Remember—each neuron is a very simple component which does nothing but executes the activation function. There are several commonly used activation functions; for example, this is the sigmoid function: f(x) = 1 / 1 + exp(-x)

The forward pass Our simple neural network works by:

1. Taking each of the two inputs
2. Multiplying by the first-layer weights—w1,2,3,4
3. Adding bias
4. Applying the activation function for neurons h1 and h2
5. Taking the output of h1 and h2, multiplying by the second layer weights—w5,6,7,8
6. This is the output.

To take a concrete example, say the first input i1 is 0.1, the weight going into the first neuron, w1, is 0.27, the second input i2 is 0.2, the weight from the second weight to the first neuron, w3, is 0.57, and the first layer bias b1 is 0.4. The input of the first neuron h1 is combined from the two inputs, i1 and i2: (i1 * w1) + (i2 * w2) + b1 = (0.1 * 0.27) + (0.2 * 0.57) + (0.4 * 1) = 0.541 Feeding this into the activation function of neuron h1: f(0.541) = 1 / (1 + exp(-0.541)) = **0.632** Now, given some other weights w2 and w4 and the second input i2, you can follow a similar calculation to get an output for the second neuron in the hidden layer, h2. The final step is to take the outputs of neurons h1 and h2, multiply them by the weights w5,6,7,8, and feed them to the same activation function of neurons o1 and o2 (exactly the same calculation as above). The result is the final output of the neural network—let's say **the final outputs are 0.735 for o1 and 0.455 for o2**. We'll also assume that the **correct output values are 0.5 for o1 and 0.5 for o2** (these are assumed correct values because in supervised learning, each data point had its truth value). **The error function** For simplicity, we'll use the Mean Squared Error function. For the first output, the error is the correct output value minus the actual output of the neural network: 0.5—0.735 = -0.235 For the second output: 0.5—0.455 = 0.045 Now we'll calculate the Mean Squared Error: MSE(o1) = ½ $(-0.235)^2$ = 0.0276 MSE(o2) = ½ $(0.045)^2$ = 0.001 The Total Error is the sum of the two errors: Total Error = 0.0276 + 0.001 = **0.0286** This is the number we need to minimize with backpropagation. **Backpropagation with gradient descent** The backpropagation algorithm calculates how much the final output values, o1 and o2, are affected by each of the weights. To do this, it calculates partial derivatives, going back from the error function to the neuron that carried a specific weight. For example, weight w6, going from hidden neuron h1 to output neuron o2, affected our model as follows: neuron h1 with weight w6 → affects total input of neuron o2 → affects output o2 → affects total errors Backpropagation goes in the opposite direction: total errors → affected by output o2 → affected by total input of neuron o2 → affected by neuron h1 with weight w6 The algorithm calculates three derivatives:

- The derivative of total errors with respect to output o2
- The derivative of output o2 with respect to total input of neuron o2
- Total input of neuron o2 with respect to neuron h1 with weight w6

This gives us complete traceability from the total errors, all the way back to the weight w6. Using the Leibniz Chain Rule, it is possible to calculate, based on the above three derivatives, what is the optimal value of w6 that minimizes the error function. **In other words, what is the "best" weight w6 that will make the neural network most accurate.** Similarly, the algorithm calculates an optimal value for each of the 8 weights. Or, in a realistic model, for each of thousands or millions of weights used for all neurons in the model. **End result of backpropagation** The backpropagation algorithm results in a set of optimal weights, like this: Optimal w1 = 0.355 Optimal w2 = 0.476 Optimal w3 = 0.233 Optimal w4 = 0.674 Optimal w5 = 0.142 Optimal w6 = 0.967 Optimal w7 = 0.319 Optimal w8 = 0.658 You can update the weights to these values, and start using the neural network to make predictions for new inputs.

How Often Are the Weights Updated?

There are three options for updating weights during backpropagation:

Updating after every sample in training set—running a forward pass for every sample, calculating optimal weights and updating. The downside is that this can be time-consuming for large training sets, and outliers can throw off the model and result in the selection of inappropriate weights.

Updating in batch—dividing training samples into several large batches, running a forward pass on all training samples in a batch, and then calculating backpropagation on all the samples together. Training is performed iteratively on each of the batches. This makes the model more resistant to outliers and variance in the training set.

Randomized mini-batches—a compromise between the first two approaches is to randomly select small batches from the training data, and run forward pass and backpropagation on each batch, iteratively. This avoids a biased selection of samples in each batch, which can lead to the of a local optimum.

Backpropagation in Real-Life Deep Learning Frameworks

In the real world, when you create and work with neural networks, you will probably not run backpropagation explicitly in your code. Deep learning frameworks have built-in implementations of backpropagation, so they will simply run it for you. Below are specifics of how to run backpropagation in two popular frameworks, Tensorflow and

Keras. To learn how to set up a neural network, perform a forward pass and explicitly run through the propagation process in your code, see Chapter 2 of Michael Nielsen's deep learning book (using Python code with the Numpy math library), or this post by Dan Aloni which shows how to do it using Tensorflow.

Tensorflow: Creating a simple neural network and running backpropagation In the code below (see the original code on StackOverflow), the line in bold performs backpropagation.

```
y0 = tf.constant( y_ , dtype=tf.float32 )  # Layer 1 = the 2x3 hidden sigmoid
m1 = tf.Variable( tf.random_uniform( [2,3] , minval=0.1 , maxval=0.9 , dtype=tf.float32  ))
b1 = tf.Variable( tf.random_uniform( [3]   , minval=0.1 , maxval=0.9 , dtype=tf.float32  ))
h1 = tf.sigmoid( tf.matmul( x0,m1 ) + b1 )  # Layer 2 = the 3x1 sigmoid output
m2 = tf.Variable( tf.random_uniform( [3,1] , minval=0.1 , maxval=0.9 , dtype=tf.float32  ))
b2 = tf.Variable( tf.random_uniform( [1]   , minval=0.1 , maxval=0.9 , dtype=tf.float32  ))
y_out = tf.sigmoid( tf.matmul( h1,m2 ) + b2 )  ### loss
# loss : sum of the squares of y0—y_out
loss = tf.reduce_sum( tf.square( y0—y_out ) )  # training step : gradient decent (1.0) to minimize loss
train = tf.train.GradientDescentOptimizer(1.0).minimize(loss)  ### training
# run 500 times using all the X and Y
# print out the loss and any other interesting info
with tf.Session() as sess:
sess.run( tf.global_variables_initializer() )
for step in range(500) :
sess.run(train)  results = sess.run([m1,b1,m2,b2,y_out,loss])
labels  = "m1,b1,m2,b2,y_out,loss".split(",")
for label,result in zip(*(labels,results)) :
print ""
print label
print result
```

Keras: Running backpropagation implicitly Keras performs backpropagation implicitly with no need for a special command. Simply create a model and train it—see the quick Keras tutorial—and as you train the model, backpropagation is run automatically.

Backpropagation and Neural Network Training in the Real World

We hope this article has helped you grasp the basics of backpropagation and neural network model training. Once you understand the mechanics, backpropagation will become something that just happens "under the hood", and your focus will shift to

running real-world models at scale, tuning hyperparameters and deriving useful results. Today's deep learning frameworks let you run models quickly and efficiently with just a few lines of code. However, in real-world projects you will run into a few challenges:

Tracking experiment progress, source code, metrics and hyperparameters across multiple experiments and training sets.

Running experiments across multiple machines—you'll need to provision these machines, configure them, and figure out how to distribute the work.

Manage training data—deep learning projects involving images or video can have training sets in the petabytes. Managing all this data, copying it to training machines and then erasing and replacing with fresh training data, can be complex and time-consuming.

Neural Networks for Image Recognition: Methods, Best Practices, Applications

missinglink.ai/guides/neural-network-concepts/neural-networks-image-recognition-methods-best-practices-applications

Image recognition has entered the mainstream and is used by thousands of companies and millions of consumers every day. Under the hood, image recognition is powered by deep learning, specifically Convolutional Neural Networks (CNN), a neural network architecture which emulates how the visual cortex breaks down and analyzes image data. **In this page you will learn about:**

What is Image Recognition?

Image recognition uses artificial intelligence technology to automatically identify objects, people, places and actions in images. Image recognition is used to perform tasks like labeling images with descriptive tags, searching for content in images, and guiding robots, autonomous vehicles, and driver assistance systems. Image recognition is natural for humans and animals but is an extremely difficult task for computers to perform. Over the past two decades, the field of Computer Vision has emerged, and tools and technologies have been developed which can rise to the challenge. The most effective tool found for the task for image recognition is a deep neural network (see our guide on artificial neural network concepts), specifically a Convolutional Neural Network (CNN). CNN is an architecture designed to efficiently process, correlate and understand the large amount of data in high-resolution images.

How Does Image Recognition Work?

The human eye sees an image as a set of signals, interpreted by the brain's visual cortex. The outcome is an experience of a scene, linked to objects and concepts that are retained in memory. Image recognition imitates this process. Computers 'see' an image as a set of vectors (color annotated polygons) or a raster (a canvas of pixels with discrete numerical values for colors). In the process of image recognition, the vector or raster encoding of the image is turned into constructs that depict physical objects and features. Computer vision systems can logically analyze these constructs, first by simplifying images and extracting the most important information, then by organizing data through feature extraction and classification. Finally, computer vision systems use classification or other algorithms to make a decision about the image or part of it – which category they belong to, or how they can best be described.

Image Recognition Algorithms

One type of image recognition algorithm is an image classifier. It takes an image (or part of an image) as an input and predicts what the image contains. The output is a class label, such as dog, cat or table. The algorithm needs to be trained to learn and

distinguish between classes. In a simple case, to create a classification algorithm that can identify images with dogs, you'll train a neural network with thousands of images of dogs, and thousands of images of backgrounds without dogs. The algorithm will learn to extract the features that identify a "dog" object and correctly classify images that contain dogs. While most image recognition algorithms are classifiers, other algorithms can be used to perform more complex activities. For example, a Recurrent Neural Network can be used to automatically write captions describing the content of an image.

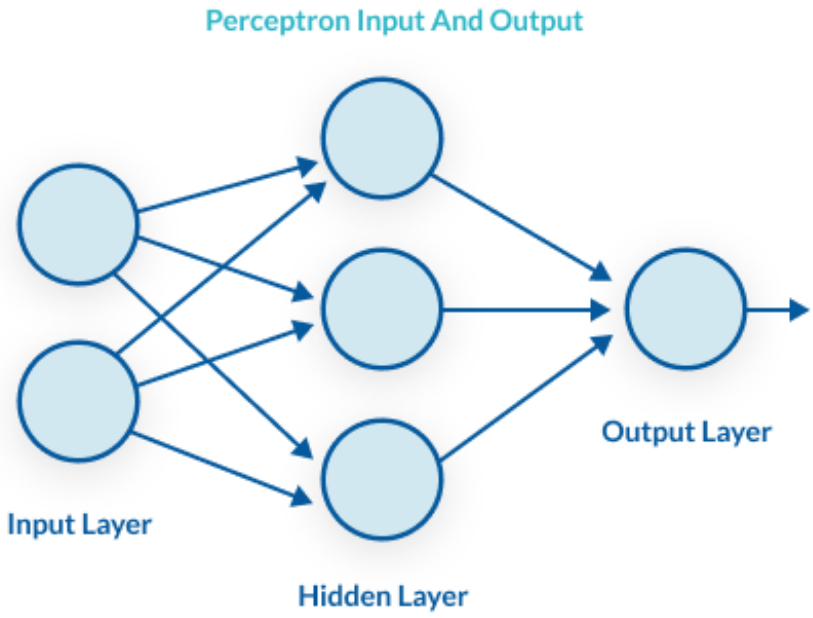
Image Data Pre-Processing Steps for Neural Networks

Neural network models for image recognition rely on the quality of the dataset – the images used to train and test the model. Here are a few important parameters and considerations for image data preparation.

- **Image size**—higher quality image give the model more information but require more neural network nodes and more computing power to process.
- **The number of images**—the more data you feed to a model, the more accurate it will be, but ensure the training set represents the real population.
- **The number of channels**—grayscale image have 2 channels (black and white) and color images typically have 3 color channels (Red, Green, Blue / RGB), with colors represented in the range [0,255].
- **Aspect ratio**—ensure the images have the same aspect ratio and size. Typically, neural network models assume a square shape input image.
- **Image scaling**—once all images are squared you can scale each image. There are many up-scaling and down-scaling techniques, which are available as functions in deep learning libraries.
- **Mean, standard deviation of input data**—you can look at the 'mean image' by calculating the mean values for each pixel, in all training examples, to obtain information on the underlying structure in the images.
- **Normalizing image inputs**—ensures that all input parameters (pixels in this case) have a uniform data distribution. This makes convergence speedier when you train the network. You can conduct data normalization by subtracting the mean from each pixel and then dividing the outcome by the standard deviation.
- **Dimensionality reduction**—you can decide to collapse the RGB channels into a gray-scale channel. You may want to reduce other dimensions if you intend to make the neural network invariant to that dimension or to make training less computationally intensive.
- **Data augmentation**—involves augmenting the existing data-set, with perturbed types of current images, including scaling and rotating. You do this to expose the neural network to a variety of variations. This way this neural network is less likely to identify unwanted characteristics in the data-set.

Building a Predictive Model for Images with Neural Networks

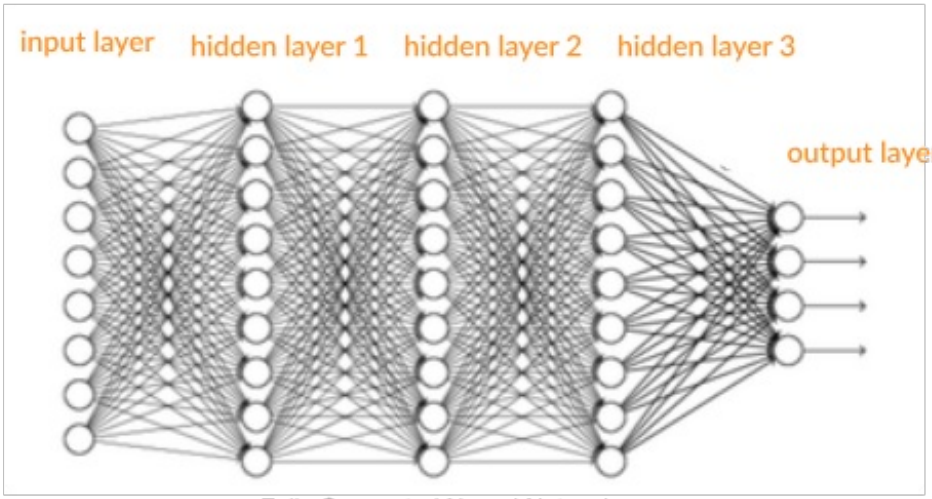
Once training images are prepared, you'll need a system that can process them and use them to make a prediction on new, unknown images. That system is an artificial neural network. Neural networks can classify just about anything, from text to images, audio files, and videos (see our in-depth article on classification and neural networks). Neural networks are an interconnected collection of nodes called neurons or perceptrons. Every neuron takes one piece of the input data, typically one pixel of the image, and applies a simple computation, called an activation function to generate a result. Each neuron has a numerical weight that affects its result. That result is fed to additional neural layers until at the end of the process the neural network generates a prediction for each input or pixel.



This process is repeated for a large number of images, and the network learns the most appropriate weights for each neuron which provide accurate predictions, in a process called backpropagation. Once a model is trained, it is applied to a new set of images which did not participate in training (a test or validation set), to test its accuracy. After some tuning, the model can be used to classify real-world images.

Limitations of Regular Neural Networks for Image Recognition

Traditional neural networks use a fully-connected architecture, as illustrated below, where every neuron in one layer connects to all the neurons in the next layer.



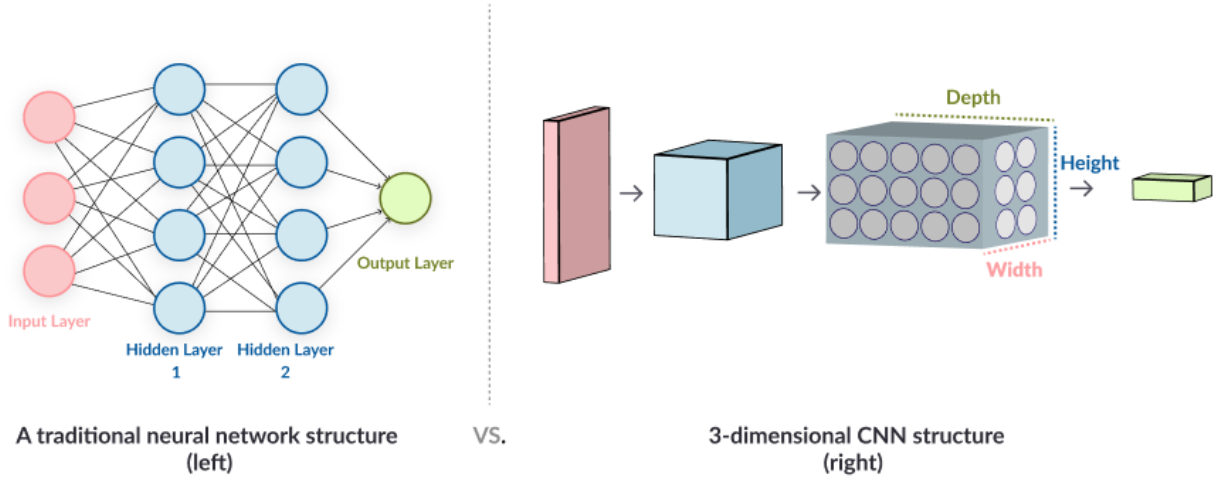
Fully-Connected Neural Network

A fully connected architecture is inefficient when it comes to processing image data:

- For an average image with hundreds of pixels and three channels, a traditional neural network will generate millions of parameters, which can lead to overfitting.
- The model would be very computationally intensive.
- It may be difficult to interpret results, debug and tune the model to improve its performance.

Convolutional Neural Networks and Their Role in Image Recognition

Unlike a fully connected neural network, in a Convolutional Neural Network (CNN) the neurons in one layer don't connect to all the neurons in the next layer. Rather, a convolutional neural network uses a three-dimensional structure, where each set of neurons analyzes a specific region or "feature" of the image. CNNs filters connections by proximity (pixels are only analyzed in relation to pixels nearby), making the training process computationally achievable. In a CNN each group of neurons focuses on one part of the image. For example, in a cat image, one group of neurons might identify the head, another the body, another the tail, etc. There may be several stages of segmentation in which the neural network analyzes smaller parts of the images, for example, within the head, the cat's nose, whiskers, ears, etc. The final output is a vector of probabilities, which predicts, for each feature in the image, how likely it is to belong to a class or category.



A traditional neural network structure (left) VS. 3-dimensional CNN structure (right)

To learn more about how CNNs work, see our in-depth Convolutional Neural Networks Guide.

Effectiveness and Limitations of Convolutional Neural Networks

A CNN architecture makes it possible to predict objects and faces in images using industry benchmark datasets with up to 95% accuracy, greater than human capabilities which stand at 94% accuracy. Even so, convolutional neural networks have their limitations:

- **Require high processing power**. Models are typically trained on high-cost machines with specialized Graphical Processing Units (GPUs).
- **Can fail when images are rotated or tilted**, or when an image has the features of the desired object, but not in the correct order or position, for example, a face with the nose and mouth switched around. A new architecture called CAPSNet has emerged to address this limitation.

Image Recognition Applications

Implementations of image recognition include security and surveillance, face recognition, visual geolocation, gesture recognition, object recognition, medical image analysis, driver assistance, and image tagging and organization in websites or large databases. Image recognition has entered the mainstream. Face, photo, and video frame recognition is used in production by Facebook, Google, Youtube, and many other high profile consumer applications. Toolkits and cloud services have emerged which can help smaller players integrate image recognition into their websites or applications.

Use of Image Recognition Across Industries

- **E-commerce industry**—image recognition is used to automatically process, categorize and tag product images, and enable powerful image search. For example, consumers can search for a chair with a particular armrest and receive relevant results.

- **Gaming industry**—image recognition can be used to transpose a digital layer on top of images from the real world. Augmented reality adds details to the existing environment. Pokemon Go is a popular game that relies on image recognition technology.
- **Automotive industry**—autonomous vehicles are in testing phases in the United States and are used for public transport in many European cities. To facilitate autonomous driving, image recognition is taught to identify objects on the road, including moving objects, vehicles, people and pathways, as well as recognize traffic lights and road signs.
- **Manufacturing**—image recognition is employed in different stages of the manufacturing cycle. It is used to reduce defects within the manufacturing process, for example, by storing images of components with related metadata and automatically identifying defects.
- **Education**—image recognition can help students with learning difficulties and disabilities. For example, applications powered by computer vision provide image-to-speech and text-to-speech functions, which can read out materials to students with dyslexia or impaired vision.

Tracking experiment source code, configuration, and hyperparameters. Convolutional networks can have many parameter and structural variations. You'll need to run hundreds or thousands of experiments to find hyperparameters that provide the best performance. Organizing, tracking and sharing experiment data and results can be a challenge.

Scaling experiments on-premise or in the cloud—CNNs are computationally intensive, and in real projects, you'll need to scale experiments across multiple machines. Provisioning machines, whether on-premise or on the cloud, setting them up to run deep learning projects and distributing experiments between them, is time-consuming.

Manage training data—computer vision projects involve rich media such as images or video, with large training sets weighing Gigabytes to Petabytes. Copying data to each training machine, then re-copying when you change training sets, can be time-consuming and error-prone.

7 Types of Neural Network Activation Functions: How to Choose?

Neural network activation functions are a crucial component of deep learning. Activation functions determine the output of a deep learning model, its accuracy, and also the computational efficiency of training a model—which can make or break a large scale neural network. Activation functions also have a major effect on the neural network's ability to converge and the convergence speed, or in some cases, activation functions might prevent neural networks from converging in the first place.

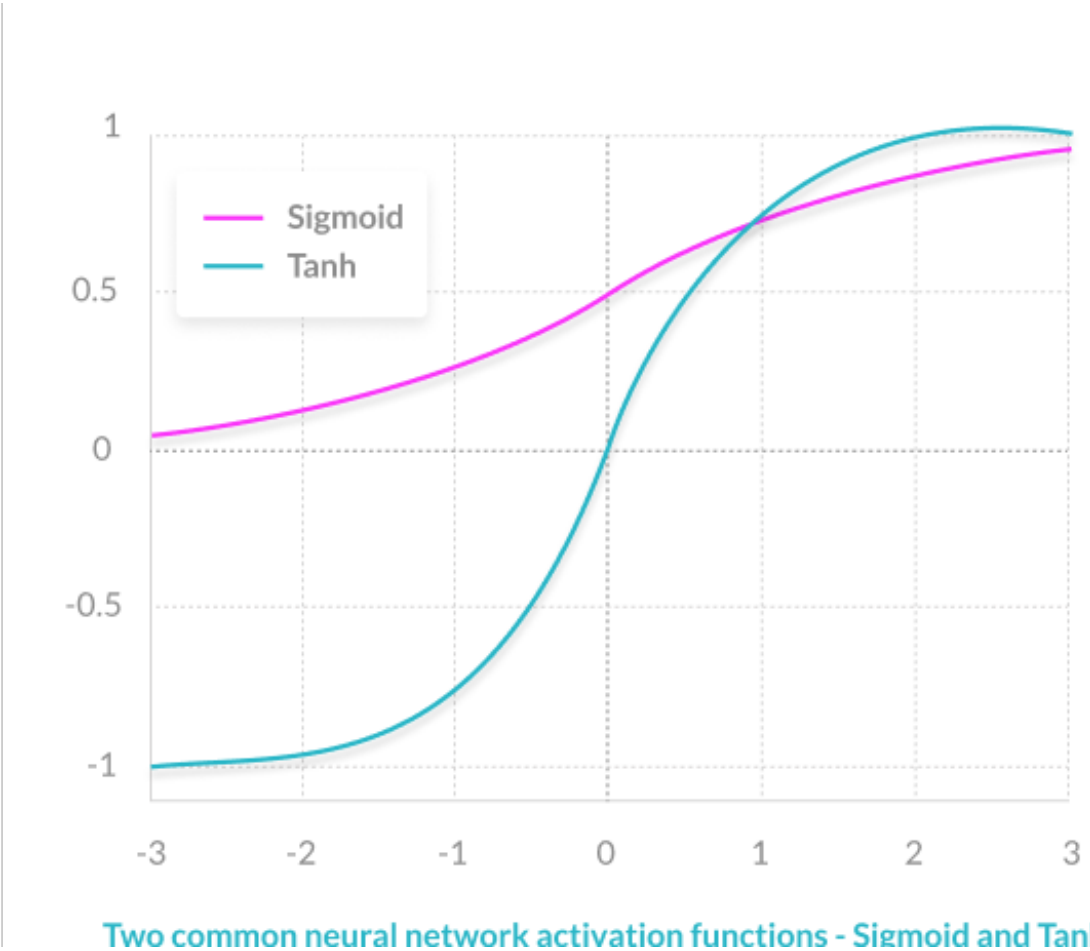
This article is part of MissingLink's Neural Network Guide, which focuses on practical explanations of concepts and processes, skipping the theoretical background. In this article you'll learn:

What is a Neural Network Activation Function?

Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction. Activation functions also help normalize the output of each neuron to a range between 1 and 0 or between -1 and 1.

An additional aspect of activation functions is that they must be computationally efficient because they are calculated across thousands or even millions of neurons for each data sample. Modern neural networks use a technique called backpropagation to train the model, which places an increased computational strain on the activation function, and its derivative function.

The need for speed has led to the development of new functions such as ReLu and Swish (see more about nonlinear activation functions below).



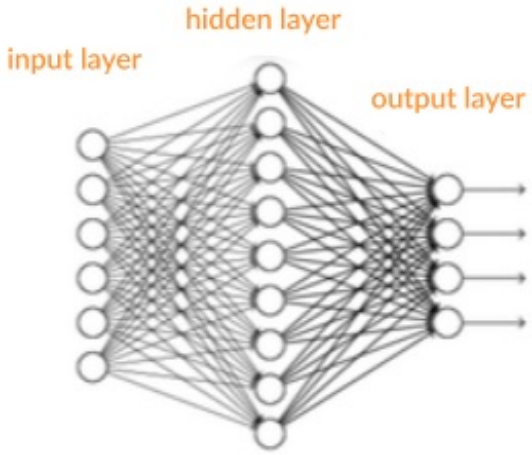
Two common neural network activation functions - Sigmoid and Tanh

What are Artificial Neural Networks and Deep Neural Networks?

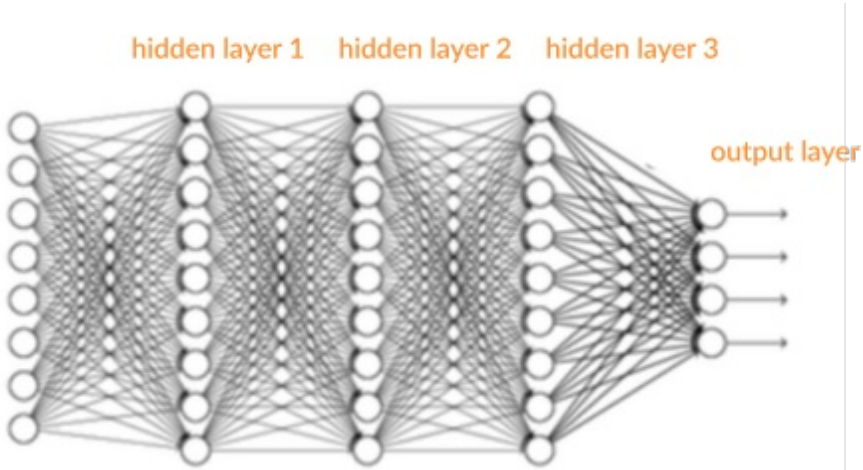
Artificial Neural Networks (ANN) are comprised of a large number of simple elements, called neurons, each of which makes simple decisions. Together, the neurons can provide accurate answers to some complex problems, such as natural language processing, computer vision, and AI.

A neural network can be “shallow”, meaning it has an input layer of neurons, only one “hidden layer” that processes the inputs, and an output layer that provides the final output of the model. A Deep Neural Network (DNN) commonly has between 2-8 additional layers of neurons. Research from Goodfellow, Bengio and Courville and other experts suggests that neural networks increase in accuracy with the number of hidden layers.

"Non-deep" feedforward neural network



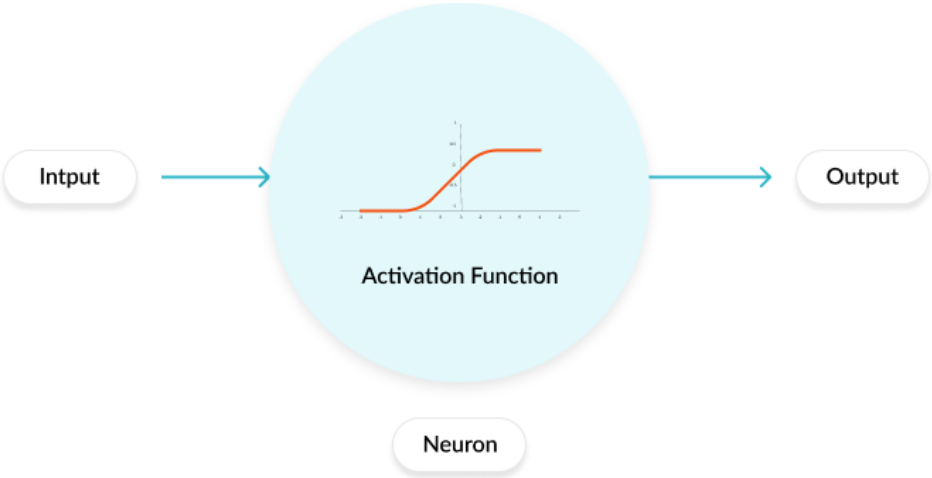
Deep neural network



Role of the Activation Function in a Neural Network Model

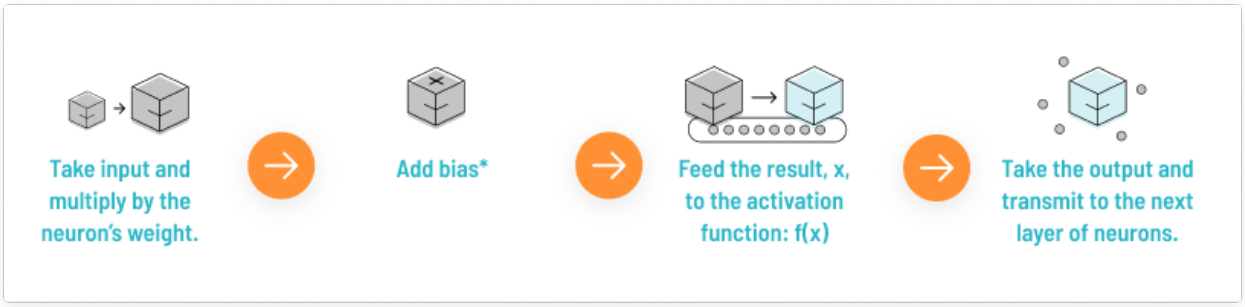
In a neural network, numeric data points, called inputs, are fed into the neurons in the input layer. Each neuron has a weight, and multiplying the input number with the weight gives the output of the neuron, which is transferred to the next layer.

The activation function is a mathematical “gate” in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold. Or it can be a transformation that maps the input signals into output signals that are needed for the neural network to function.



Increasingly, neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.

The basic process carried out by a neuron in a neural network is:

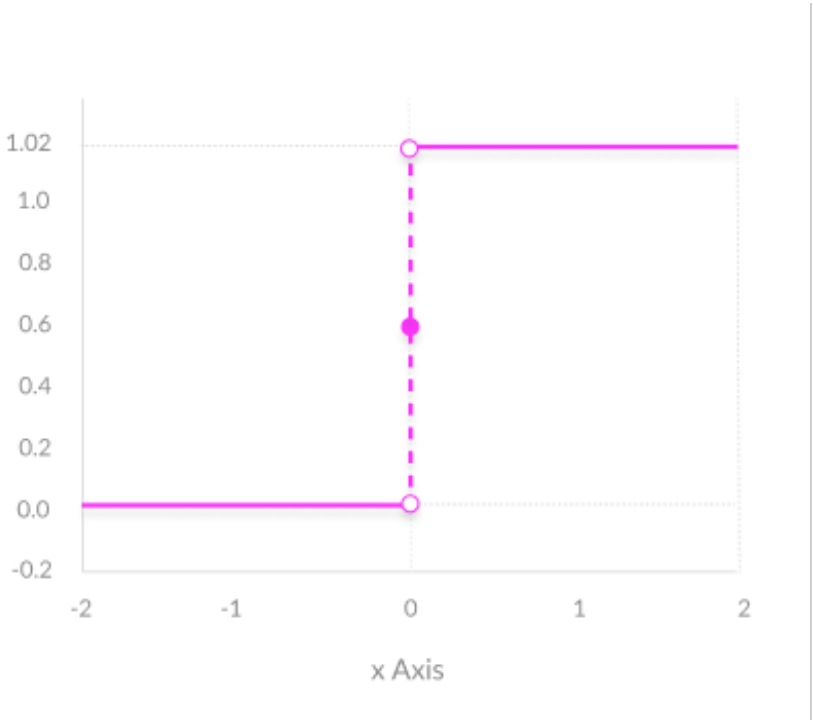


* This is just the number 1, making it possible to represent activation functions that do not cross the origin. Biases are also assigned a weight.

3 Types of Activation Functions

Binary Step Function

A binary step function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated and sends exactly the same signal to the next layer.

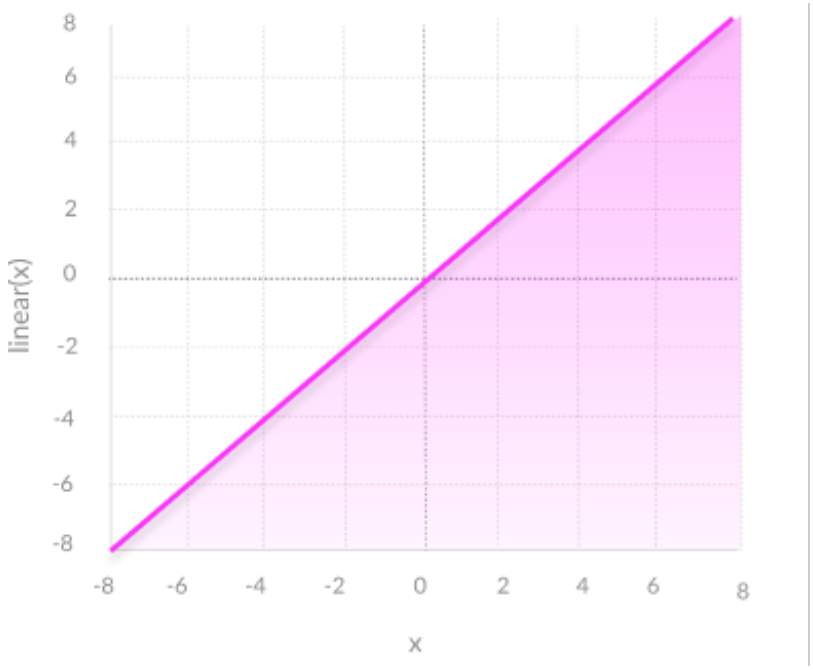


The problem with a step function is that it does not allow multi-value outputs—for example, it cannot support classifying the inputs into one of several categories.

Linear Activation Function

A linear activation function takes the form:

A = cx



It takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input. In one sense, a linear function is better than a step function because it allows multiple outputs, not just yes and no.

However, a linear activation function has two major problems:

1. Not possible to use backpropagation (gradient descent) to train the model—the derivative of the function is a constant, and has no relation to the input, X. So it's not possible to go back and understand which weights in the input neurons can provide a better prediction.

Go in-depth: See our guide on backpropagation

2. All layers of the neural network collapse into one—with linear activation functions, no matter how many layers in the neural network, the last layer will be a linear function of the first layer (because a linear combination of linear functions is still a linear function). So a linear activation function turns the neural network into just one layer.

A neural network with a linear activation function is simply a linear regression model. It has limited power and ability to handle complexity varying parameters of input data.

Non-Linear Activation Functions

Modern neural network models use non-linear activation functions. They allow the model to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality.

Almost any process imaginable can be represented as a functional computation in a neural network, provided that the activation function is non-linear.

Non-linear functions address the problems of a linear activation function:

1. They allow backpropagation because they have a derivative function which is related to the inputs.
2. They allow "stacking" of multiple layers of neurons to create a deep neural network. Multiple hidden layers of neurons are needed to learn complex data sets with high levels of accuracy.

7 Common Nonlinear Activation Functions and How to Choose an Activation Function

Sigmoid / Logistic

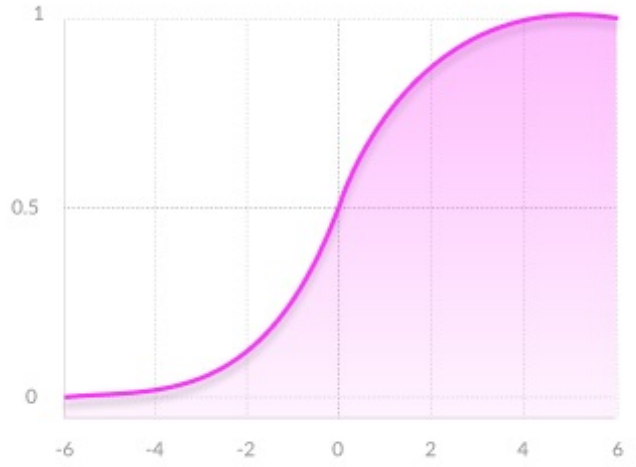
Advantages

- **Smooth gradient**, preventing "jumps" in output values.
- **Output values bound** between 0 and 1, normalizing the output of each neuron.
- **Clear predictions**—For X above 2 or below -2, tends to bring the Y value (the

prediction) to the edge of the curve, very close to 1 or 0. This enables clear predictions.

Disadvantages

- **Vanishing gradient**—for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
- **Outputs not zero centered**.
- **Computationally expensive**



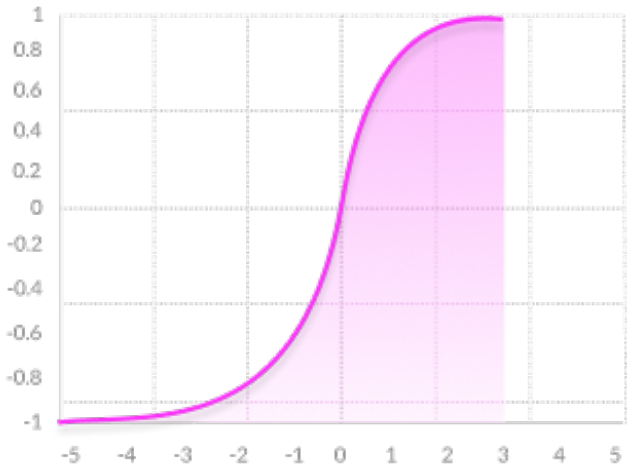
TanH / Hyperbolic Tangent

Advantages

- **Zero centered**—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.
- Otherwise like the Sigmoid function.

Disadvantages

Like the Sigmoid function



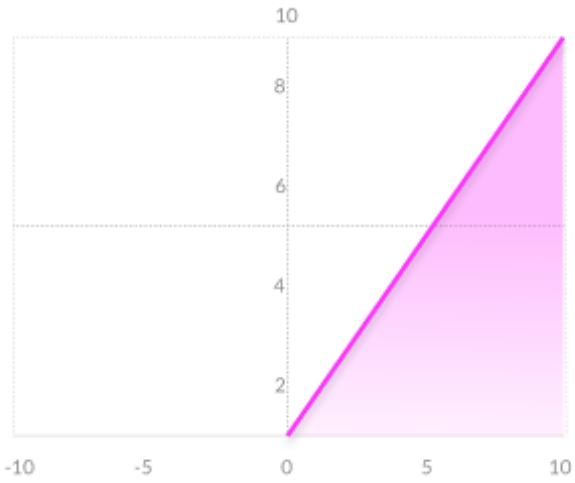
ReLU (Rectified Linear Unit)

Advantages

- **Computationally efficient**—allows the network to converge very quickly
- **Non-linear—**although it looks like a linear function, ReLU has a derivative function and allows for backpropagation

Disadvantages

The Dying ReLU problem—when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.



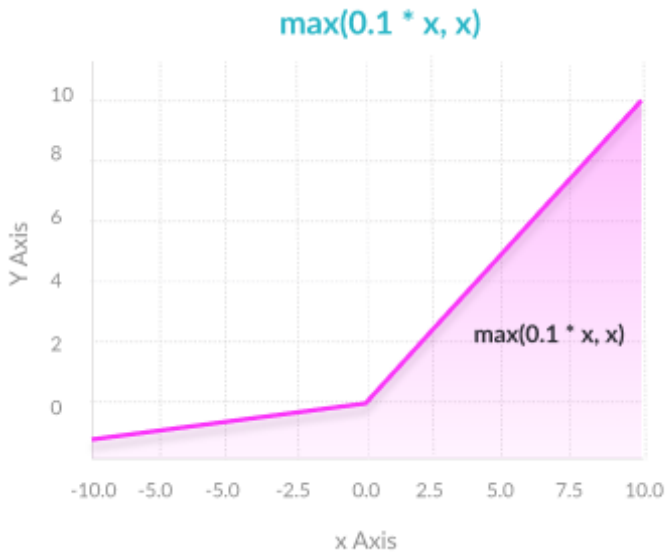
Leaky ReLU

Advantages

- **Prevents dying ReLU problem**—this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values
- Otherwise like ReLU

Disadvantages

Results not consistent—leaky ReLU does not provide consistent predictions for negative input values.



Parametric ReLU

Advantages

- **Allows the negative slope to be learned**—unlike leaky ReLU, this function provides the slope of the negative part of the function as an argument. It is, therefore, possible to perform backpropagation and learn the most appropriate value of α.
- Otherwise like ReLU

Disadvantages

May perform differently for different problems.

$$f(x)=max(\alpha\partial,\partial)$$

Softmax

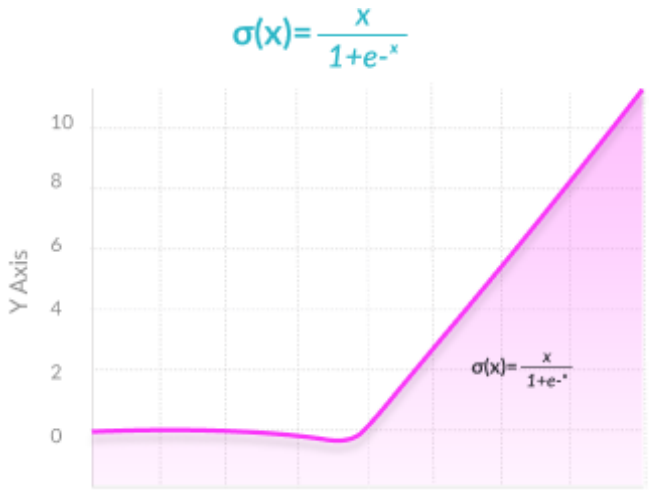
Advantages

- **Able to handle multiple classes** only one class in other activation functions—normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class.
- **Useful for output neurons**—typically Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories.

$$\sigma(z)j=\frac{ezj}{\sum_{k=1}^{k}ezk}\quad for\ j=1....k.$$

Swish

Swish is a new, self-gated activation function discovered by researchers at Google. According to their paper, it performs better than ReLU with a similar level of computational efficiency. In experiments on ImageNet with identical models running ReLU and Swish, the new function achieved top -1 classification accuracy 0.6-0.9% higher.



Derivatives or Gradients of Activation Functions

The derivative—also known as a gradient—of an activation function is extremely important for training the neural network.

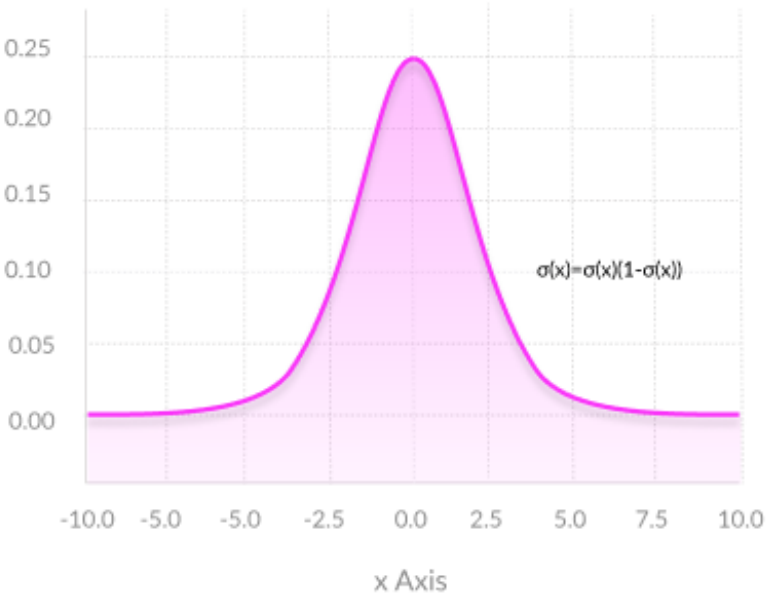
Neural networks are trained using a process called backpropagation—this is an algorithm which traces back from the output of the model, through the different neurons which were involved in generating that output, back to the original weight applied to each neuron. Backpropagation suggests an optimal weight for each neuron which results in the most accurate prediction.

Go in-depth: See our guide on backpropagation

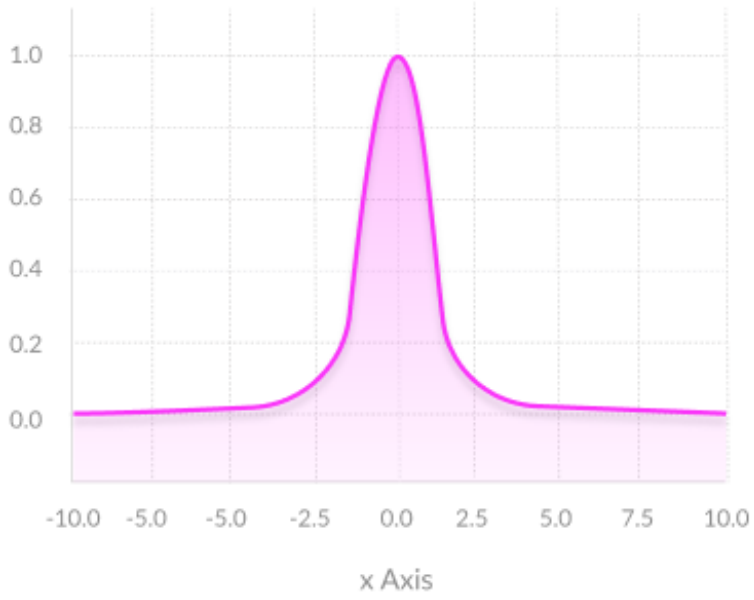
Below are the derivatives for the most common activation functions.

Activation Functions and their Derivative Graph (used for backpropagation):

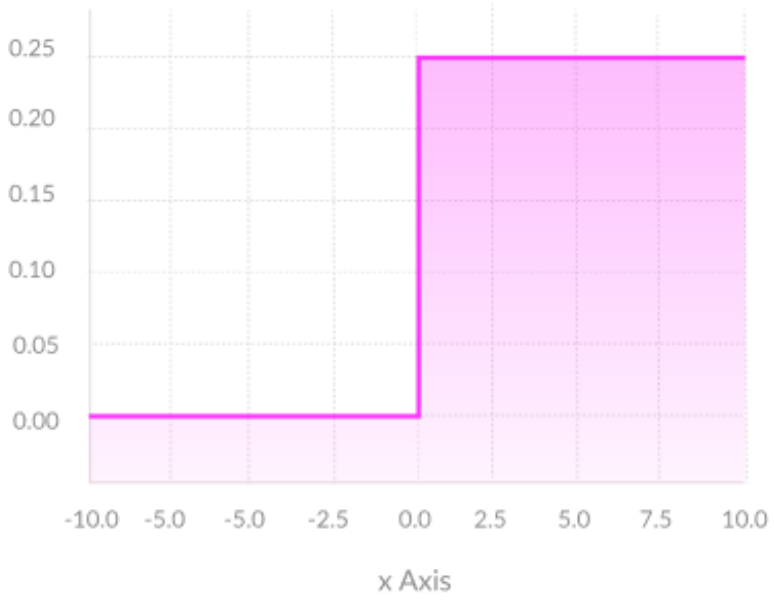
Sigmoid


0.25
0.20
0.15
0.10
0.05
0.00
σ(x)=σ(x)(1-σ(x))
-10.0 -5.0 -5.0 -2.5 0.0 2.5 5.0 7.5 10.0
x Axis

TanH


1.0
0.8
0.6
0.4
0.2
0.0
-10.0 -5.0 -5.0 -2.5 0.0 2.5 5.0 7.5 10.0
x Axis

ReLU



Recent research by Franco Manessi and Alessandro Rozza attempted to find ways to automatically learn which is the optimal activation function for a certain neural network and to even automatically combine activation functions to achieve the highest accuracy. This is a very promising field of research because it attempts to discover an optimal activation function configuration automatically, whereas today, this parameter is manually tuned.

Neural Network Activation Functions in the Real World

When building a model and training a neural network, the selection of activation functions is critical. Experimenting with different activation functions for different problems will allow you to achieve much better results.

In a real-world neural network project, you will switch between activation functions using the deep learning framework of your choice.

For example, here is how to use the ReLU activation function via the Keras library (see all supported activations):

```
keras.activations.relu(x, alpha=0.0, max_value=None)
```

While selecting and switching activation functions in deep learning frameworks is easy, you will find that managing multiple experiments and trying different activation functions on large test data sets can be challenging.

It can be difficult to:

Track experiment progress source code, metrics and hyperparameters across different experiments trying different activation functions for a model, or variations of the same model.

Run experiments across multiple machines running multiple large scale experiments will usually require you to run on several machines; you'll need to provision and maintain these machines.

Manage training data to achieve good results, you'll need to experiment with different sets of test data across multiple model variations on different machines. Moving the training data each time you need to run an experiment is difficult, especially if you are processing heavy inputs like images or video.

Convolutional Neural Network: How to Build One in Keras & PyTorch

missinglink.ai/guides/neural-network-concepts/convolutional-neural-network-build-one-keras-pytorch

A lot has been written about CNNs in theory. How do you build them in practice? This article will show the commands to build CNN in two popular DL frameworks, and short tutorials to get you started.

In this article you will learn:

A Brief Intro to Convolutional Neural Networks

Convolutional Neural Networks (CNN) have proven very good at processing data that is closely knitted together. A CNN uses a three-dimensional structure, with three specialized neural networks analyzing the red, green and blue layers of a color image. CNN scans an image one area at a time, identifies and extracts important features, and uses them to classify the image.

CNNs are primarily used for computer vision, powering tasks like image classification, face recognition, identifying and classifying everyday objects, and image processing in robots and autonomous vehicles. They are also used for video analysis and classification, semantic parsing, automatic caption generation, search query retrieval, sentence classification, and more.

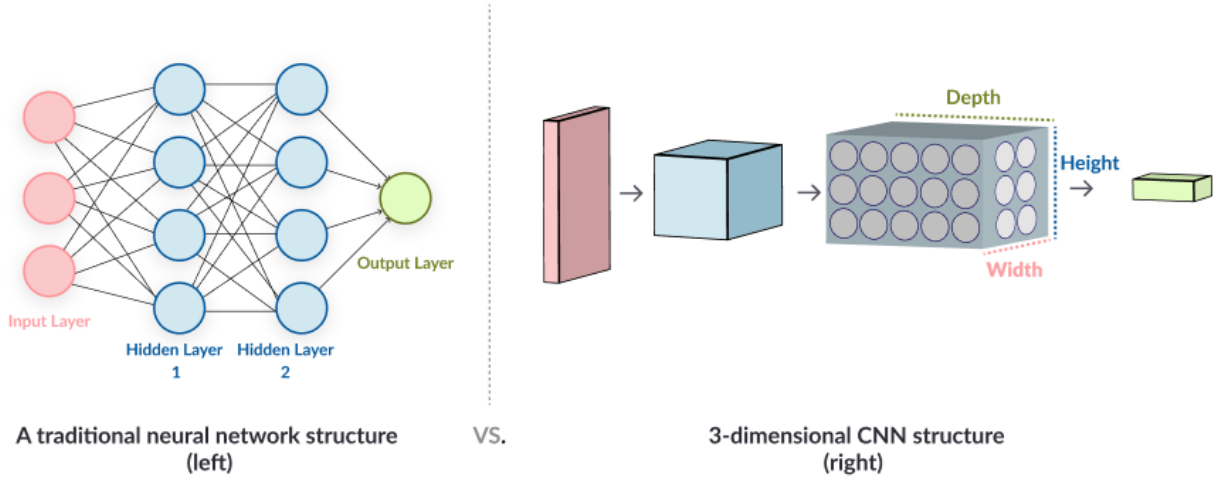
While the typical CNN uses 2-dimensional or 3-dimensional neural layers to analyze images with 2 or 3 color channels, CNNs with 1-dimensional layers are also very useful. A 1D CNN can derive important features from short segments of an overall dataset when the position of each segment is not so important. For example, sensor data, audio signals, and natural language processing.

CNN Architecture

A plain vanilla neural network, in which all neurons in one layer communicate with all the neurons in the next layer (this is called “fully connected”), is inefficient when it comes to analyzing large images and video. For an average size image with hundreds of pixels and three color channels (red, green, blue), the number of parameters using a traditional neural network will be in the millions, which can lead to overfitting.

To limit the number of parameters and focus the neural network on important parts of the image, a CNN uses a three-dimensional structure in which each set of neurons analyzes a small region or “feature” of the image. Instead of having all neurons pass their decisions to the next neural layer, each group of neurons specializes on identifying one

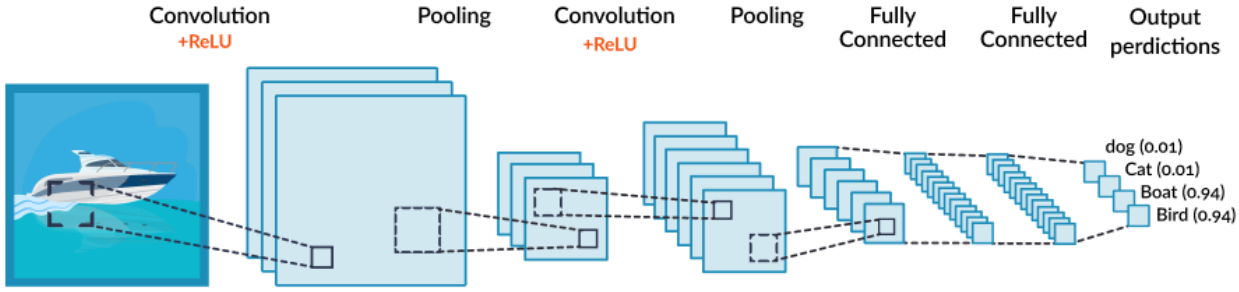
part of the image, for example, the nose, left ear, mouth or hair. The final output is a vector of probability scores, representing how likely each of the features is to be part of a class.



A traditional neural network structure (left) VS. 3-dimensional CNN structure (right)

How CNNs Works

A CNN operates in three stages. The first is a *convolution*, in which the image is "scanned" a few pixels at a time, and a *feature map* is created with probabilities that each feature belongs to the required class (in a simple classification example). The second stage is *pooling* (also called downsampling), which reduces the dimensionality of each feature while maintaining its most important information. The pooling stage creates a "summary" of the most important features in the image.



Most CNNs use "max pooling", in which the highest value is taken from each pixel area scanned by the CNN, as illustrated below.

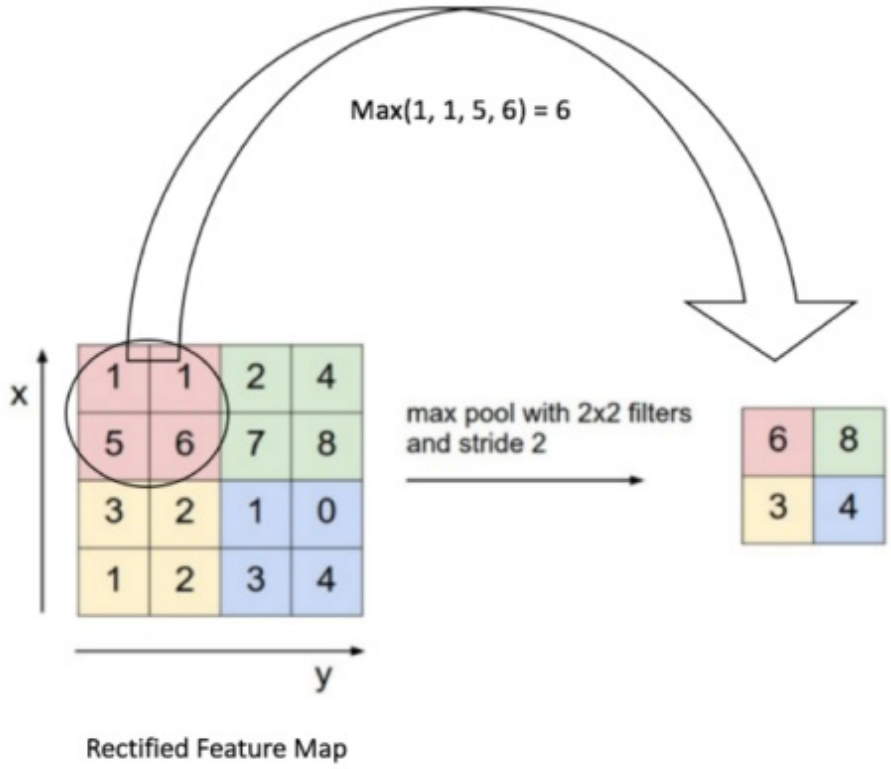


Image Source: Stanford

A CNN can perform several rounds of convolution then pooling. For example, in the first round, an image can be broken down into objects, such as a boat, a person, a plane of grass. In the second round, the CNN can identify features within each object, for example, a face, torso, hands, legs. In a third round, the CNN could go deeper and analyze features within the face, etc.

Finally, when the features are at the right level of granularity, the CNN enters the third stage, which is a fully-connected neural network that analyzes the final probabilities, and decides which class the image belongs to. The final step can also be used for other tasks, such as generating text—a common use of convolutional networks is to automatically generate captions for images.

We deliberately kept the theoretical part short, to get to the punch of how to build a CNN yourself.

> *To understand convolutional neural networks in more detail, see this in-depth guide from Andrej Karpathy*

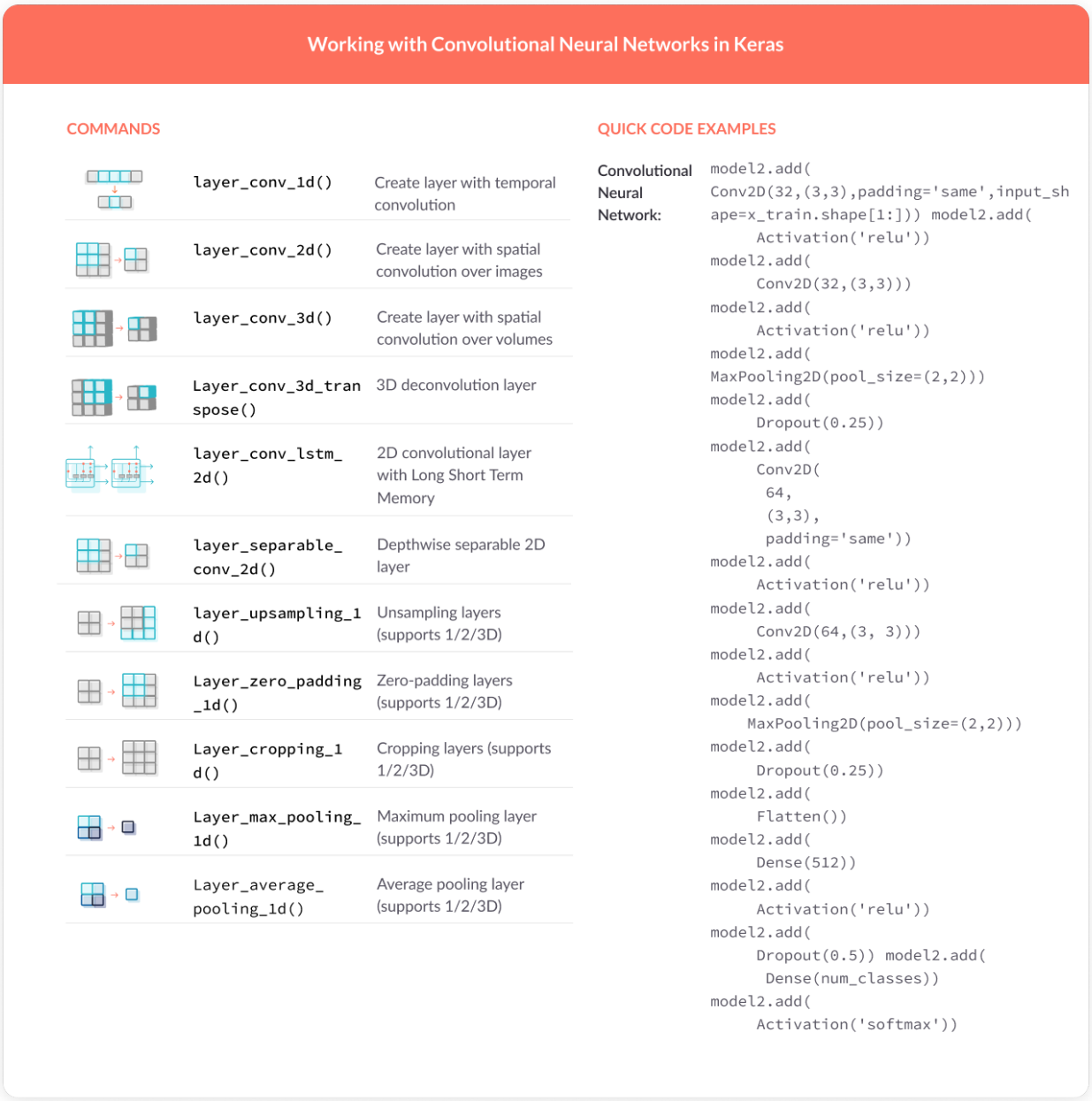
Enough talk—let's see how it's done.

Your First Convolutional Neural Network in Keras

Keras is a high-level deep learning framework which runs on top of TensorFlow, Microsoft Cognitive Toolkit or Theano. It lets you build standard neural network structures with only a few lines of code. To customize and create your own deep learning algorithms, you'll need to work directly with TensorFlow or another lower-level deep learning framework.

CNN in Keras is based on a sequential model—you define parameters, create a model object and add convolutional layers to it.

Keras CNN Commands Cheat Sheet



Working with Convolutional Neural Networks in Keras

COMMANDS

`layer_conv_1d()`	Create layer with temporal convolution
`layer_conv_2d()`	Create layer with spatial convolution over images
`layer_conv_3d()`	Create layer with spatial convolution over volumes
`Layer_conv_3d_transpose()`	3D deconvolution layer
`layer_conv_lstm_2d()`	2D convolutional layer with Long Short Term Memory
`layer_separable_conv_2d()`	Depthwise separable 2D layer
`layer_upsampling_1d()`	Unsampling layers (supports 1/2/3D)
`Layer_zero_padding_1d()`	Zero-padding layers (supports 1/2/3D)
`Layer_cropping_1d()`	Cropping layers (supports 1/2/3D)
`Layer_max_pooling_1d()`	Maximum pooling layer (supports 1/2/3D)
`Layer_average_pooling_1d()`	Average pooling layer (supports 1/2/3D)

QUICK CODE EXAMPLES

Convolutional Neural Network:

```
model2.add(
Conv2D(32,(3,3),padding='same',input_shape=x_train.shape[1:])) model2.add(
     Activation('relu'))
model2.add(
     Conv2D(32,(3,3)))
model2.add(
     Activation('relu'))
model2.add(
MaxPooling2D(pool_size=(2,2)))
model2.add(
     Dropout(0.25))
model2.add(
     Conv2D(
      64,
      (3,3),
      padding='same'))
model2.add(
     Activation('relu'))
model2.add(
     Conv2D(64,(3, 3)))
model2.add(
     Activation('relu'))
model2.add(
    MaxPooling2D(pool_size=(2,2)))
model2.add(
     Dropout(0.25))
model2.add(
     Flatten())
model2.add(
     Dense(512))
model2.add(
     Activation('relu'))
model2.add(
     Dropout(0.5)) model2.add(
      Dense(num_classes))
model2.add(
     Activation('softmax'))
```

Training a CNN on the MNIST Dataset in Keras—a Brief Tutorial

This tutorial will show you how to load the MNIST dataset and, a benchmark deep learning dataset, containing 70,000 handwritten numbers from 0-9, and building a convolutional neural network to classify the handwritten digits. Our discussion is based on the excellent tutorial by Elijaz Allibhai.

Follow these steps to train CNN on MNIST and generate predictions:

1. Load the MNIST dataset and split into train and test sets, with X_train and X_test containing the training and testing images, and `y_train` and `y_test` containing the "ground truth" of the digits represented in the images. In the MNIST data set 60,000 images are used for training and 10,000 for testing/validation (learn more about neural network bias and variance in our neural network guide).

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

2. View one of the images using the `plt.imshow` command, and check its size using the `.shape` function, to understand what the dataset looks like. Try this on several images. As you will see, all the MNIST images are uniformly 28 x 28 pixels in size and contain handwritten digits.

```
import matplotlib.pyplot as plt
plt.imshow(X_train[0])
X_train[0].shape
```

3. Reshape the two sets of images, `X_train` and `X_test`, to the shape expected by the CNN model. The Keras `reshape` function takes four arguments: number of training images, pixel size, and image depth—use 1 to indicate a grayscale image.

```
X_train = X_train.reshape(60000,28,28,1)
X_test = X_test.reshape(10000,28,28,1)
```

4. Next, you'll need to 'one-hot-encode' the target variable—create a column for each classification category, with each column containing binary values indicating if the current image belongs to that category or not. Because we are classifying digits, there will be 10 columns for digits 0-9, and according to the classification decision, one of the columns will have a 1 (e.g. the column for the digit 3) and the rest will be 0.

Note: If you train with normal class numbers, this will introduce a bias. The model will consider higher value digits (e.g., 9) to be greater than lower value digits (e.g. 1). One-hot-encoding causes the model to treat all digits as equivalent.

```
from keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
y_train[0]
```

5. Create a model, using the `Sequential` model type, which lets you build a model by adding on one layer at a time.

```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
model = Sequential()
```

6. Add model layers: the first two layers are Conv2D—2-dimensional convolutional layers These are convolution layers that deal with the input images, which are seen as 2-dimensional matrices. The `Conv2D` function takes four parameters:

- Number of neural nodes in each layer. We will use 64 for the first convolutional layer and 32 for the second.
- `kernel_size` defines the filter size—this is the area in square pixels the model will use to "scan" the image. Kernel size of 3 means the model looks at a square of 3×3 pixels at a time.
- activation is the type of activation function (click to learn more in our neural network guide) we use after each convolutional layer. For CNN the typical activation function used is ReLu.
- `input_shape` is the pixel size of the images and the image depth, again setting 1 for grayscale.

```
model.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
```

Note: Each of the convolution layers reduces the depth, width and height of each feature, this is equivalent to the pooling/downsampling stage in the CNN model. The formula for calculating the output size for any given conv layer is:
O = ((W-K+2P)/S)+1 where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

7. Add a 'Flatten' layer, which takes the output of the two convolution layers and turns it into a format that can be used by the final, densely connected neural layer.

```
model.add(Flatten())
```

8. Add the final layer of type 'Dense', a densely-connected neural layer which will generate the final prediction. The Dense function takes two arguments:

- Number of output nodes—10 in our case because we need to generate predictions for digits between 0-9.
- Type of activation function for the output layer. We use `softmax` which is the typical activation function used for neural output layers. Softmax takes the Dense layer output and converts it to meaningful probabilities for each of the digits, which sum up to 1. It then makes a prediction based on the digit that has the highest probability.

```
model.add(Dense(10, activation='softmax'))
```

9. Compile the model. The `compile` function takes three parameters:

- `optimizer` controls the learning rate, which defines how fast optimal weights for the model are calculated (learn more about hyperparameters in our neural network guide). We will use the 'adam' learning rate optimizer.
- `loss` defines the loss function, which measures how far the model's prediction is from the ground truth, the correct digits for the images (learn more about loss functions and the backpropagation process). We will use 'categorical_crossentropy', a loss function suitable for classification problems.
- `metrics` defines how we evaluate model success. We'll use the 'accuracy' metric to calculate an accuracy score on the testing/validation set of images.

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

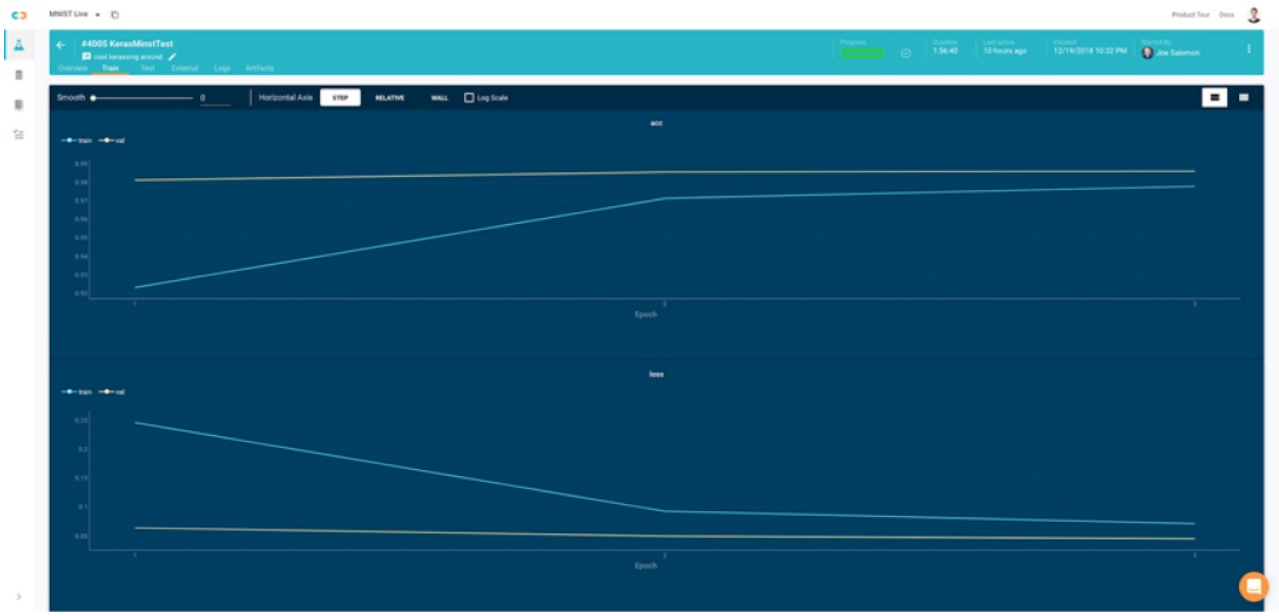
10. Training the model, using the `fit` function which takes three parameters:

- training data (train_X)
- target data (train_y)
- validation data
- number of epochs—number of times the backpropagation process will be run on the training images—we will set this to 3.

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3)
```

11. The model trains, the output should look like this:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 22s 363us/step - loss: 1.3991 - acc: 0.8830 - val_loss: 0.0882 - val_acc: 0.9738
Epoch 2/3
60000/60000 [==============================] - 20s 334us/step - loss: 0.0712 - acc: 0.9790 - val_loss: 0.0874 - val_acc: 0.9729
Epoch 3/3
60000/60000 [==============================] - 20s 334us/step - loss: 0.0484 - acc: 0.9854 - val_loss: 0.0898 - val_acc: 0.9757
<keras.callbacks.History at 0x7fc38442e240>
```



As you can see above, after three epochs, accuracy on the validation set is up to 97.57%. The CNN works well.

Now that you see the model is working, you can generate actual predictions using the `predict` function. The function returns an array of probabilities for each of the 10 possible results (digits 0-9), with the sum of probabilities for each image equal to 1. You can input new, unknown data to the predict function to get a prediction for this data. For now, let's run a prediction for the first four images in the test set:

```
model.predict(X_test[:4])
```

The output will show probabilities for digits 0-9, for each of the 4 images. The model predicts 7, 2, 1 and 0 for the first four images.

13. Compare this with actual results for the first 4 images in the test set:

```
y_test[:4]
```

The output shows that the ground truth for the .first four images is also 7, 2,1 and 0—the model made an accurate prediction.

Your First Convolutional Neural Network in PyTorch

PyTorch is a middle ground between Keras and Tensorflow—it offers some high-level commands which let you easily construct basic neural network structures. At the same time, it lets you work directly with tensors and perform advanced customization of neural network architecture and hyperparameters.

To create a CNN model in PyTorch, you use the `nn.Module` class which contains a complete neural network toolkit, including convolutional, pooling and fully connected layers for your CNN model. PyTorch lets you define parameters at every stage—dataset loading, CNN layer construction, training, forward pass, backpropagation, and model testing.

PyTorch CNN Commands Cheat Sheet

Working with Convolutional Neural Networks in PyTorch

COMMANDS

`nn.Linear(m,n)`	Create fully connected layer from m to n units
`nn.ConvXd(m,n,s)`	Create X-dimensional convolutional layer from m to n channels with kernel size s
`nn.ACTIVATION_FUNCTION_COMMAND`	ACTIVATION_FUNCTION_COMMAND can be, for example: Sigmoid, LogSigmoid, PReLU, ReLU, ReLU6, RReLU, TanH, Softmax (see all)
`nn.LOSS_FUNCTION_COMMAND`	LOSS_FUNCTION_COMMAND can be, for example: BCELoss, CrossEntropyLoss, L1Loss, MSELoss, NLLLoss, SoftMarginLoss, CosinEmbeddingLoss, KLDivLoss (see all)
`Opt = optimX( model.parameters(),...)`	Create optimizer
`opt.step()`	Update weights
`optim.OPTIMIZER_COMMAND`	OPTIMIZER_COMMAND can be, for example: SGD, Adadelta, Adagrad, Adam, Adamax, ASGD, LBFGS, RMSProp (see all)
`Scheduler = optim.X(optimizer,...)`	Create learning rate scheduler
`scheduler.step()`	Update learning rate at start of epoch
`Optim.lr_scheduler. SCHEDULER_COMMAND`	SCHEDULER_COMMAND can be, for example: LambdaLR, StepLR, MultiStepLR, ExponentialLR (see all)

CODE EXAMPLES

Create basic CNN

```
Create linear neural network on MNIST data
class LinearNetMNIST(nn.Module):
    def __init__(self):
        super(LinearNetMNIST,
self).__init__()
        # Input size (bs, 784)
        self.layer1 = nn.Linear(784, 500)
        self.layer2 = nn.Linear(500, 250)
        self.layer3 = nn.Linear(250, 10)
        self.softmax = nn.Softmax()

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.softmax(x)
        return x
Create basic CNN
class BasicCNN(nn.Module):
    def __init__(self, logger=None):
        super(BasicCNN, self).__init__()
        self.logger = logger
        self.conv1 = nn.Conv2d(in_channels=3,
out_channels=50,
                               kernel_size=2,
stride=1, padding=0)
        self.conv2 =
nn.Conv2d(in_channels=50, out_channels=100,
                               kernel_size=2,
stride=1, padding=0)
        self.linear =
nn.Linear(in_features=100*30*30,
out_features=10)
        self.softmax = nn.Softmax()

    def log(self, msg):
        if self.logger:
            self.logger.debug(msg)

    def forward(self, x):
        self.log(x.size()) # (bs,3,32,32)
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1) #
(bs,100*30*30)
        x = self.linear(x)
        x = self.softmax(x)
        return x

# Test
logger = get_logger(logging.DEBUG,
logging.DEBUG)
net = BasicCNN(logger)
inputs,targets =
next(iter(cifar_train_loader))
inputs = Variable(inputs[:2])
net(inputs)
```

Training CNN on MNIST Dataset in PyTorch

This brief tutorial shows how to load the MNIST dataset into PyTorch, train and run a CNN model on it. As mentioned above, MNIST is a standard deep learning dataset

containing 70,000 handwritten digits from 0-9. Our discussion is based on the great tutorial by Andy Thomas.

Follow these steps to train CNN on MNIST and generate predictions:

1. Set hyperparameters—these are safe to start with.

```
num_epochs = 5
num_classes = 10
batch_size = 100
learning_rate = 0.001DATA_PATH = 'C:\\...\PycharmProjects\MNISTData'MODEL_STORE_PATH =
'C:\\...\PycharmProjects\pytorch_models\\'
```

2. Specify local drive folders to store the MNIST dataset, and a location for the trained data.

```
MODEL_STORE_PATH = 'C:\\...\PycharmProjects\pytorch_models\\'
```

3. The MNIST dataset comes built into PyTorch, accessible via `torchvision.datasets.MNIST`. Transform the dataset using the `transforms.Compose()` function, as follows:

- Convert the input data set to a PyTorch tensor.
- Normalize the data, supplying the mean (0.1307) and standard deviation (0.3081) of the MNIST dataset. You need to do this for every channel in the dataset, but because MNIST is grayscale, there is only one channel and one mean/STD pair.

```
trans = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
```

4. Create the `train_dataset` and `test_dataset` objects, providing the following arguments:

- `root` —specifies the folder where the train.pt and test.pt data files exist
- `train` —specifies whether to use the train.pt or test.pt data file
- `transform` —passes the transform object, created earlier
- `download` —specifies that the MNIST data should be downloaded from source if needed

```
train_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=True, transform=trans,
download=True)
test_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=False, transform=trans)
```

5. Load the train and test datasets into the data loader. A data loader can be used as an iterator – to extract the data, just use a standard Python iterator such as `enumerate`. The `DataLoader` function takes three arguments:

- The data set you wish to load
- The batch size
- Whether you wish to randomly shuffle the data

```
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

6. Create the CNN model by initializing the `nn.Module` class. This is a PyTorch class which has everything you need to build a neural network. It also provides recursive operations, ways of parallelizing work and moving it to a GPU or back to a CPU, and more. We'll create the following neural layers:

- `layer1` —using the `nn.Sequential` object, we create a compound layer that includes a 2D convolutional layer, a ReLu activation function and a 2D MaxPool layer.
    - The Conv2d method lets you define, in this order: number of input channels (1 because MNIST images are grayscale), number of output channels, the size of the convolutional filter (you can supply a tuple for different shapes), stride and padding.
    - Stride and padding for the convolutional layer are defined using this equation: Wout=(Win–F+2P)S+1 where Wout=width of output, Win = width of input, F=filter, S=stride, P=padding. The same formula can be used for height, since the images are symmetric. Because we want the output size to be the same as the input (2), we set stride to 1 and padding to 2 → the output width and height = 2.
    - For the pooling layer, we set stride to 2 and padding to zero, to down-sample and reduce images by a factor of 2.
    - Thus, the output from `layer1` will be 32 channels of 14 x 14 pixel images.
- `layer2` —like 1, except input channels are 32 because it received the output of the first layer, and output 64 channels.
- `drop_out` layer to avoid over-fitting in the model.
- `fc1` and `fc2` —Two fully connected layers, created using the `nn.Linear` method. The first with 7 x 7 x 64 nodes, and the second with 1000 nodes. The second argument of `nn.Linear` specifies the number of nodes in the next layer.

```
class ConvNet(nn.Module):
def __init__(self):
super(ConvNet, self).__init__()
self.layer1 = nn.Sequential(
nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2))
self.layer2 = nn.Sequential(
nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2))
self.drop_out = nn.Dropout()
self.fc1 = nn.Linear(7 * 7 * 64, 1000)
self.fc2 = nn.Linear(1000, 10)
```

7. Define the forward pass. To customize forward pass functionality, call this function "forward", to override the base forward function in `nn.Module`. The second argument x

is one batch of data, which is fed into the first neural layer (`layer1`), then to the next layer, and so on. After `layer2` , we reshape the data, flattening it from 7 x 7 x 64 into 3164 x 1.

```
def forward(self, x):
out = self.layer1(x)
out = self.layer2(out)
out = out.reshape(out.size(0), -1)
out = self.drop_out(out)
out = self.fc1(out)
out = self.fc2(out)
return out
```

8. Define training parameters by creating a `ConvNet` object and defining:

- `criterion` —the loss function. We use the PyTorch `CrossEntropyLoss` function which combines a SoftMax and cross-entropy loss function.
- `optimizer` —we use the Adam optimizer, passing all the parameters from the CNN model we defined earlier, and a learning rate.

```
model = ConvNet()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

9. Train the model, by running two loops:

- Loop over the number of epochs.
- Within this loop, iterate over `train_loader` using `enumerate` , and do the following:
    - Perform a forward pass, by passing a batch of normalized MNIST images from `train_loader` to the `model` object you defined earlier. There is no need to explicitly run the forward function, PyTorch does this automatically when it executes a model.
    - Pass the `outputs` true image `labels` to the loss function.
    - Append the loss to a list, which you can use later to plot training progress.
    - In preparation for backpropagation, set gradients to zero by calling `zero_grad()` on the optimizer.
    - Perform backpropagation using the `backward()` method of the loss object. Gradients are calculated.
    - Call `optimizer.step()` to perform Adam optimizer training.

```
total_step = len(train_loader)
loss_list = []
acc_list = []for epoch in range(num_epochs):for i, (images, labels) in enumerate(train_loader):outputs
= model(images)loss = criterion(outputs,
labels)loss_list.append(loss.item())optimizer.zero_grad()loss.backward()optimizer.step()
```

10. Track accuracy (within the same loop) by:

- Running the `torch.max()` function, which returns the index of the maximum value in a tensor.

- In the first argument of the `max` function, pass a tensor of outputs from the model, which should be of size (batch_size, 10). For each sample in the batch, this will return the maximum value over the 10 output nodes, each representing one of the digits 0-9. For example, output 2 corresponds to digit "2". The node with the highest output value will be predicted by the model.
- In the second argument of the `max()` function, pass 1. This instructs the max function to examine the output node axis (axis=0 corresponds to the batch_size dimension).
- `max()` returns a list of prediction integers from the model
- Compare predictions with the true labels (`predicted == labels`) and sum them to get the number of correct predictions. Divide by the `batch_size` to obtain the accuracy.

```
total = labels.size(0)
_, predicted = torch.max(outputs.data, 1)
correct = (predicted == labels).sum().item()
acc_list.append(correct / total)if (i + 1) % 100 == 0:print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f},
Accuracy: {:.2f}%'.format(epoch + 1, num_epochs, i + 1, total_step, loss.item(),(correct / total) *
100))# output will look like this:# Epoch [1/6], Step [100/600], Loss: 0.2183, Accuracy: 95.00%
```

11. To test the model, do the following:

- Run `model.eval()` —this runs the model while disabling drop-out or batch normalization layers.
- `torch.no_grad()` disables autograd functionality in the model, this is PyTorch's mechanism for performing backpropagation and calculating gradients, which is not needed in model testing.
- The remaining code is the same as in the accuracy calculation above, except you are iterating through `test_loader` and not `train_loader` .
- Output the prediction to the console, save model results using `torch.save()` . This enables graphing the results using a plotting library.

```
model.eval()
with torch.no_grad():
correct = 0
total = 0
for images, labels in test_loader:
outputs = model(images)
_, predicted = torch.max(outputs.data, 1)
total += labels.size(0)
correct += (predicted == labels).sum().item()print('Test Accuracy of the model on the 10000 test
images: {} %'.format((correct / total) * 100))torch.save(model.state_dict(), MODEL_STORE_PATH +
'conv_net_model.ckpt')
```

Convolutional Neural Networks in the Real World

In this article, we explained the basics of Convolutional Neural Networks and showed how to create them in two popular deep learning frameworks, Keras and PyTorch. When you start working on CNN projects, processing and generating predictions for real

images, audio and video, you'll run into some practical challenges:

Tracking experiment progress, source code, and hyperparameters across multiple CNN experiments. Convolutional networks can have many structural variations and hyperparameter tweaks, and testing each of these on your data will require running an experiment and tracking its results.

Running experiments across multiple machines—CNNs are very computationally intensive, and real projects will likely require running experiments on multiple machines or GPUs. Provisioning these machines, configuring them and distributing the work among them can become a burden.

Manage training data—CNN projects commonly involve rich media such as images or video, and training sets can weight anywhere from Gigabytes to Petabytes. Copying data to training machines and replacing it for different experiments can be time-consuming, and even infeasible in extreme cases.

Neural Networks for Regression (Part 1)—Overkill or Opportunity?

missinglink.ai/guides/neural-network-concepts/neural-networks-regression-part-1-overkill-opportunity

Regression models have been around for many years and have proven very useful in modeling real world problems and providing useful predictions, both in scientific and in industry and business environments. In parallel, neural networks and deep learning are growing in adoption, and are able to model complex problems and provide predictions that resemble the learning process of the human brain. What's the connection between neural networks and regression problems? Can you use a neural network to run a regression? Is there any benefit to doing so? The short answer is yes—because most regression models will not perfectly fit the data at hand. If you need a more complex model, applying a neural network to the problem can provide much more prediction power compared to a traditional regression. **In Part 1 of this article you will learn:**

In Part 2 of this article [coming soon]:

- How to run neural networks mimicking regression models in Tensorflow
- How to run neural network regressions in Keras

What is Regression Analysis?

Regression analysis can help you model the relationship between a dependent variable (which you are trying to predict) and one or more independent variables (the input of the model). Regression analysis can show if there is a significant relationship between the independent variables and the dependent variable, and the strength of the impact—when the independent variables move, by how much you can expect the dependent variable to move. **The simplest, linear regression equation looks like this:**

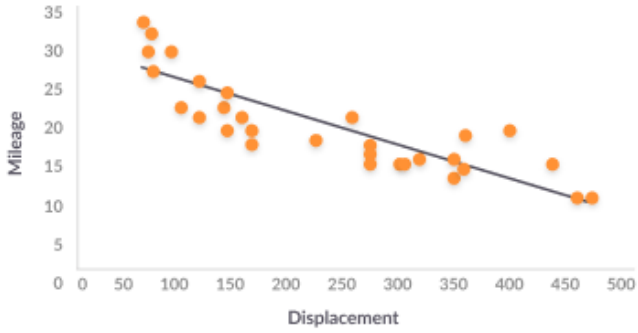
$$\gamma = \beta_1 + \beta_2 X_2 + \beta_3 X_3 + \cdots + \beta_k X_k + \varepsilon$$

- y → dependent variable—the value the regression model is aiming to predict
- X2,3..k → independent variables—one or more values that the model takes as an input, using them to predict the dependent variables
- [beta]1,2,3..k → Coefficients—these are weights that define how important each of the variables is for predicting the dependent variable
- [error] → Error—the distance between the value predicted by the model and the actual dependent variable y. Statistical methods can be used to estimate and reduce the size of the error term, to improve the predictive power of the model.

Types of Regression Analysis

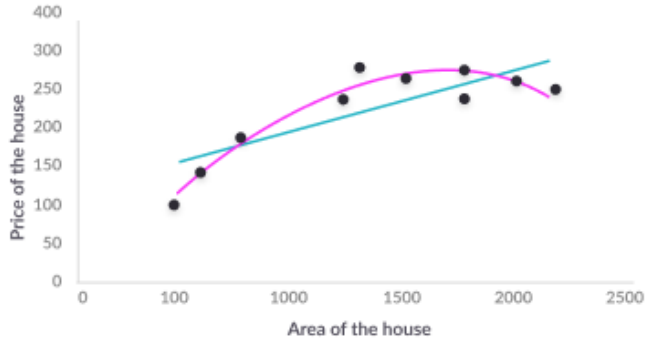
Linear Regression

Suitable for dependent variables which are continuous and can be fitted with a linear function (straight line).



Polynomial Regression

Suitable for dependent variables which are best fitted by a curve or a series of curves. Polynomial models are prone to overfitting, so it is important to remove outliers which can distort the prediction curve.



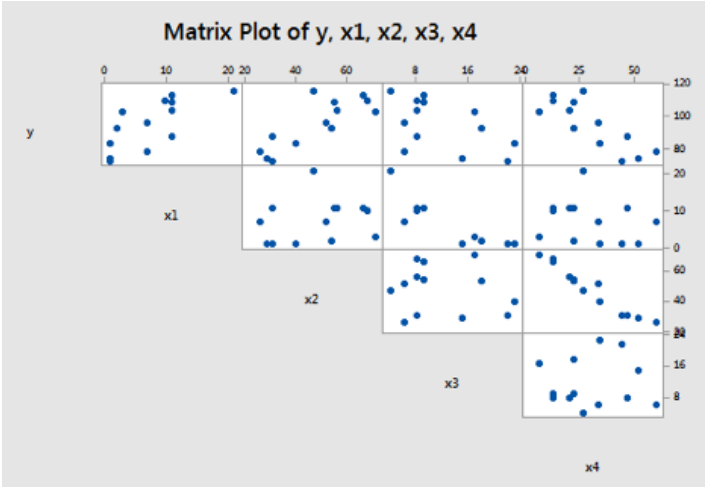
Logistic Regression

Suitable for dependent variables which are binary. Binary variables are not normally distributed—they follow a binomial distribution, and cannot be fitted with a linear regression function.

$$Y_i = \frac{1}{1-e^{-X_i\beta}} + \varepsilon_i \text{ where } -\infty < x < \infty, y = 0,1$$

Stepwise Regression

An automated regression technique that can deal with high dimensionality—a large number of independent variables. Stepwise regression observes statistical values to detect which variables are significant, and drops or adds co-variates one by one to see which combination of variables maximizes prediction power. Image source: Penn State University



Ridge Regression

A regression technique that can help with multicollinearity—independent variables that are highly correlated, making variances large and causing a large deviation in the predicted value. Ridge regression adds a bias to the regression estimate, reducing or "penalizing' the coefficients using a shrinkage parameter. Ridge regression shrinks coefficients using least squares, meaning that the coefficients cannot reach zero. Ridge regression is a form of regularization—it uses L2 regularization (learn about bias in neural networks in our guide).

$$\text{Min } (\sum \varepsilon^2 + \lambda \sum \beta^2) = \text{Min } \sum (y - (\beta_1 + \beta_2 X_2 + \beta_3 X_3 + \ldots + \beta_k X_k))^2 + \lambda \sum \beta^2$$

Lasso Regression

Lasso Regression

Least Absolute Shrinkage and Selection Operator (LASSO) regression, similar to ridge regression, shrinks the regression coefficients to solve the multicollinearity problem. However, Lasso regression shrinks the absolute values, not the least squares, meaning some of the coefficients can become zero. This leads to “feature selection”—if a group of dependent variables are highly correlated, it picks one and shrinks the others to zero. Lasso regression is also a type of regularization—it uses L1 regularization.

$$\text{Min}\ (\sum \varepsilon^2 + \lambda \sum |\beta|) = \text{Min}\ \sum (y - (\beta_1 + \beta_2 X_2 + \beta_3 X_3 + \ldots + \beta_k X_k))^2 + \lambda \sum |\beta|$$

ElasticNet Regression

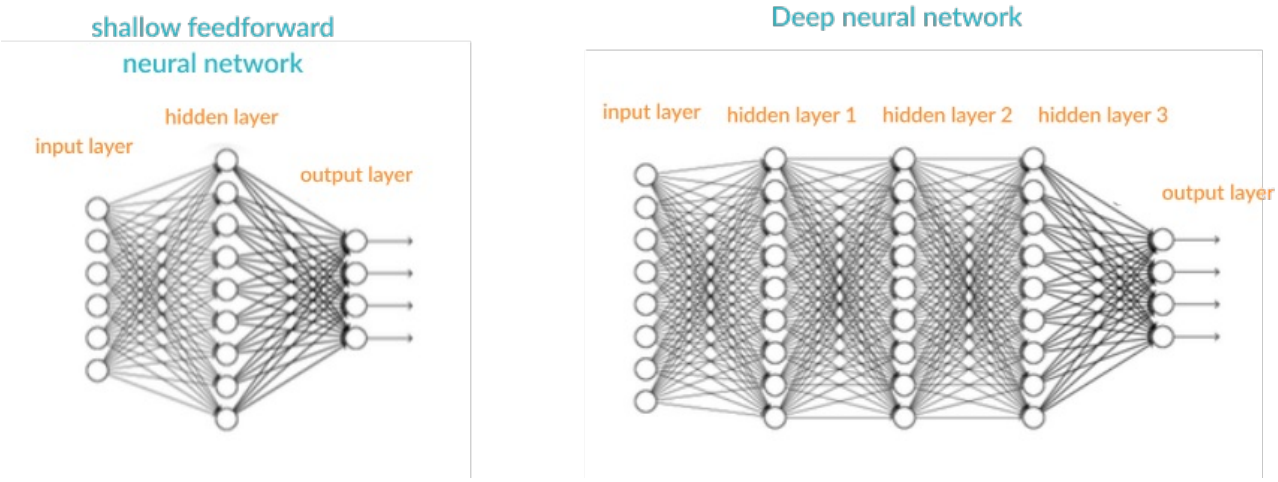
ElasticNet combines Ridge and Lasso regression, and is trained successively with L1 and L2 regularization, thus trading-off between the two techniques. The advantage is that ElasticNet gains the stability of Ridge regression while allowing feature selection like Lasso. Whereas Lasso will pick only one variable of a group of correlated variables, ElasticNet encourages a group effect and may pick more than one correlated variables.

$$\text{Min}\ (\sum \varepsilon^2 + \lambda_1 \sum \beta^2 + \lambda_2 \sum |\beta|) = \text{Min}\ \sum (y - (\beta_1 + \beta_2 X_2 + \beta_3 X_3 + \ldots + \beta_k X_k))^2 \lambda_1 \sum \beta^2 + \lambda_2 \sum |\beta|$$

What Is a Neural Network?

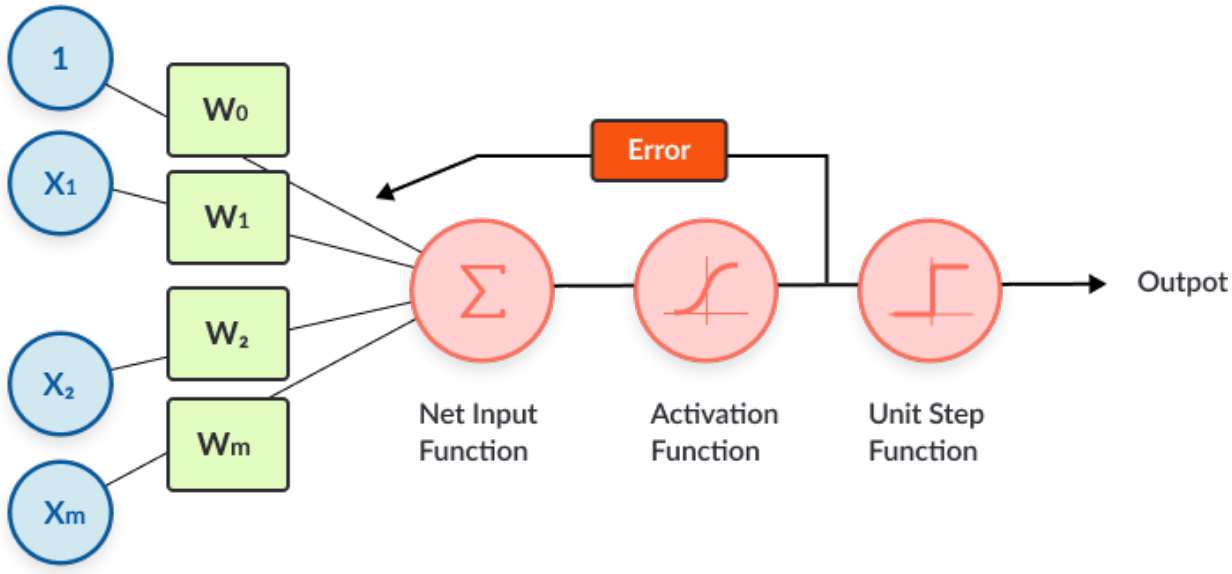
Artificial Neural Networks (ANN) are comprised of simple elements, called neurons, each of which can make simple mathematical decisions. Together, the neurons can analyze complex problems, emulate almost any function including very complex ones, and provide accurate answers. A shallow neural network has three layers of neurons: an input layer, a hidden layer, and an output layer. A Deep Neural Network (DNN) has more than one hidden layers, which increases the complexity of the model and can significantly improve prediction power.

Shallow vs deep neural networks



Regression in Neural Networks

Neural networks are reducible to regression models—a neural network can "pretend" to be any type of regression model. **For example, this very simple neural network, with only one input neuron, one hidden neuron, and one output neuron, is equivalent to a logistic regression**. It takes several dependent variables = input parameters, multiplies them by their coefficients = weights, and runs them through a sigmoid activation function and a unit step function, which closely resembles the logistic regression function with its error term.

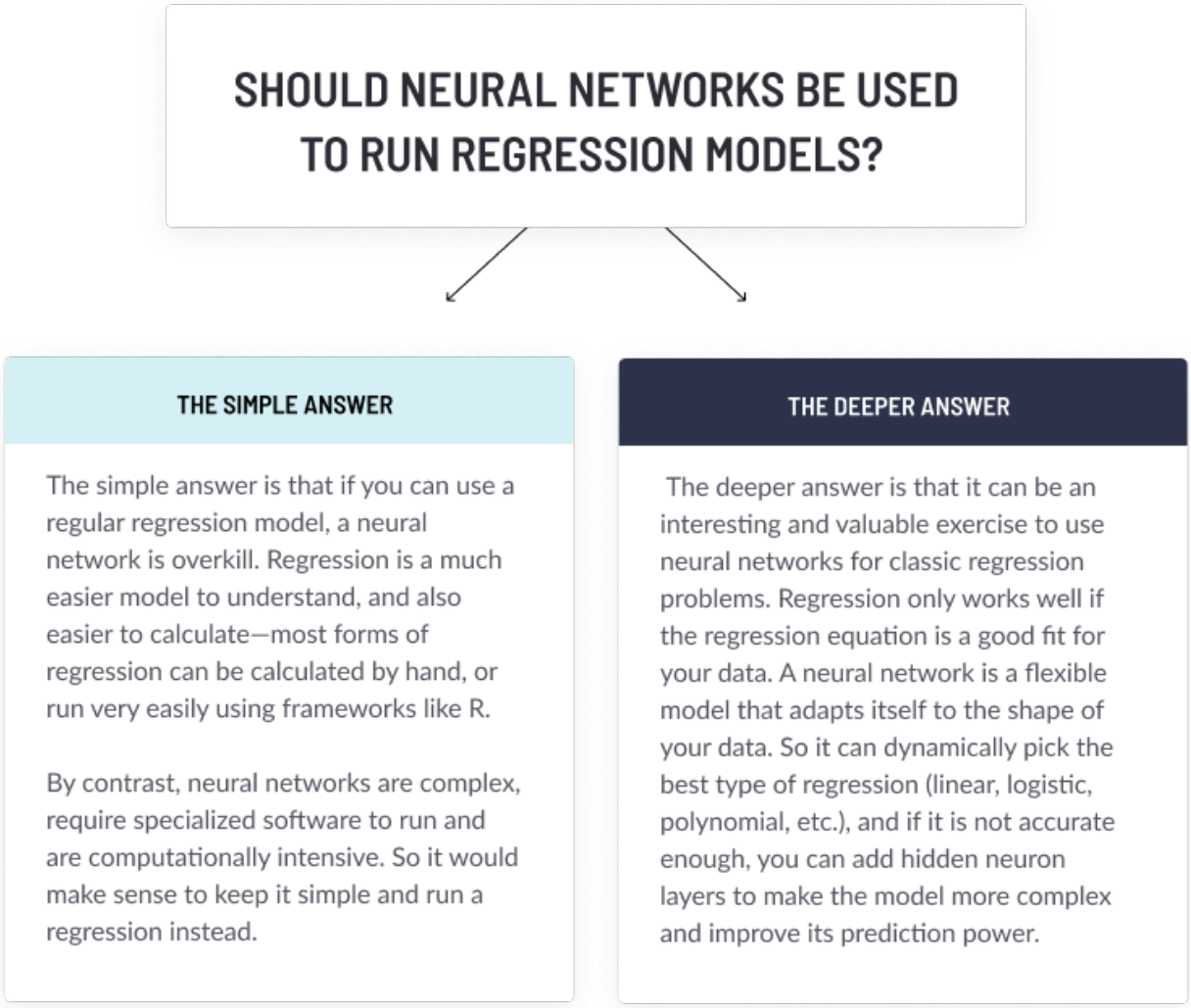


Regression in neural networks

When this neural network is trained, it will perform gradient descent (to learn more see our in-depth guide on **backpropagation**) to find coefficients that are better and fit the data, until it arrives at the optimal linear regression coefficients (or, in neural network terms, the optimal weights for the model).

A More Complex Model

The logistic regression we modeled above is suitable for binary classification. What if we need to model multi-class classification? We can increase the complexity of the model by using multiple neurons in the hidden layer, to achieve one-vs-all classification. Each classification option can be encoded using three binary digits, as shown below. For the output of the neural network, we can use the Softmax activation function (see our complete guide on **neural network activation functions**). The Softmax calculation can include a normalization term, ensuring the probabilities predicted by the model are "meaningful" (sum up to 1). This illustrates how a neural network can not only simulate a regression function, but can also model more complex scenarios by increasing the number of neurons, layers, and modifying other hyperparameters (see our complete guide on **neural network hyperparameters**).



To summarize, if a regression model perfectly fits your problem, don't bother with neural networks. But if you are modeling a complex data set and feel you need more prediction power, give deep learning a try. Chances are that a neural network can automatically construct a prediction function that will eclipse the prediction power of your traditional regression model. ***Stay tuned for part 2 of this article*** *which will show how to run regression models in Tensorflow and Keras, leveraging the power of the neural network to improve prediction power.*

Regression with Neural Networks in Real Life

Running traditional regression functions is typically done in R or other math or statistics libraries. To run a neural network model equivalent to a regression function, you will need to use a deep learning framework such as TensorFlow, Keras or Caffe, which has a steeper learning curve. As we hinted in the article, while neural networks have their overhead and are a bit more difficult to understand, they provide prediction power uncomparable to even the most sophisticated regression models. It's extremely rare to see a regression equation that perfectly fits all expected data sets, and the more complex your scenario, the more value you'll derive from "crossing the Rubicon" to the land of deep learning. When you get your start in deep learning, you'll find that with only a basic understanding of neural network concepts, the frameworks will do all the work for you. Specify the parameters and they'll build your neural network, run your experiments and deliver results. However, as you scale up your deep learning work, you'll discover additional challenges:

Tracking progress across multiple experiments and storing source code, metrics and hyperparameters. Neural networks require constant trial and error to get the model right and it's easy to get lost among hundreds or thousands of experiments.

Running experiments across multiple machines—unlike regression models, neural networks are computationally intensive. You'll quickly find yourself having to provision additional machines, as you won't be able to run large scale experiments on your development laptop. Managing those machines can be a pain.

Manage training data—depending on the project, training data can get big. If you're processing images, video or large quantities of unstructured data, managing this data and copying it to the machines that run the experiments can become difficult.

Classification with Neural Networks: Is it the Right Choice?

missinglink.ai/guides/neural-network-concepts/classification-neural-networks-neural-network-right-choice

There are many effective ways to automatically classify entities. In this article, we cover 6 common classification algorithms, of which neural networks are just one choice. Which algorithm is the best choice for your classification problem, and are neural networks worth the effort? **In part 1 of this article you will learn:**

In part 2 of this article, coming soon:

- **Coding up a neural network classifier**
- **Text classification using deep learning**
- **Image classification with Convolutional Neural Networks (CNN)**

What Is Classification in Machine and Deep Learning?

Classification involves predicting which class an item belongs to. Some classifiers are binary, resulting in a yes/no decision. Others are multi-class, able to categorize an item into one of several categories. Classification is a very common use case of machine learning—classification algorithms are used to solve problems like email spam filtering, document categorization, speech recognition, image recognition, and handwriting recognition. In this context, a neural network is one of several machine learning algorithms that can help solve classification problems. Its unique strength is its ability to dynamically create complex prediction functions, and emulate human thinking, in a way that no other algorithm can. There are many classification problems for which neural networks have yielded the best results.

Types of Classification Algorithms: Which Is Right for Your Problem?

To understand classification with neural networks, it's essential to learn how other classification algorithms work, and their unique strengths. For many problems, a neural network may be unsuitable or "overkill". For others, it might be the only solution.

Logistic Regression

Classifier type

Binary

How It Works

Analyzes a set of data points with one or more independent variables (input variables, which may affect the outcome) and finds the best fitting model to describe the data points, using the logistic regression equation:

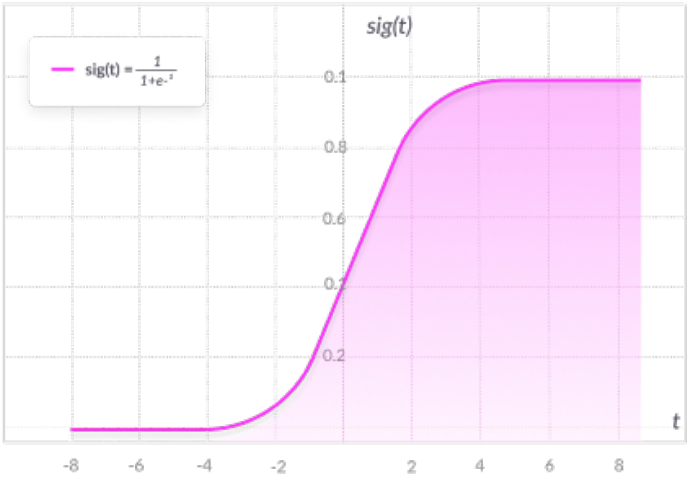
$$Y_i = \frac{1}{1-e^{-X_i'\beta}} + \varepsilon_i \text{ where } -\infty < x < \infty, y = 0,1$$

Strengths

Simple to implement and understand, very effective for problems in which the set of input variables is well known and closely correlated with the outcome.

Weaknesses

Less effective when some of the input variables are not known, or when there are complex relationships between the input variables.



Classification with Neural Networks - Binary

Decision Tree Algorithm

Classifier type

Multiclass

How it works

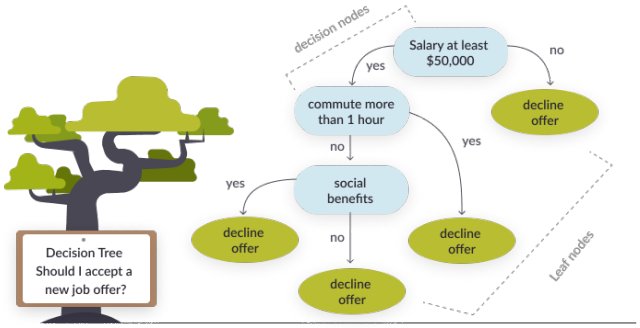
Uses a tree structure with a set of “if-then” rules to classify data points. The rules are learned sequentially from the training data. The tree is constructed top-down; attributes at the top of the tree have a larger impact on the classification decision. The training process continues until it meets a termination condition.

Strengths

Able to model complex decision processes, very intuitive interpretation of results.

Weaknesses

Can very easily overfit the data, by over-growing a tree with branches that reflect outliers in the data set. A way to deal with overfitting is pruning the model, either by preventing it from growing superfluous branches (pre-pruning), or removing them after the tree is grown (post-pruning).



Random Forest Algorithm

Classifier type

Multiclass

How it works

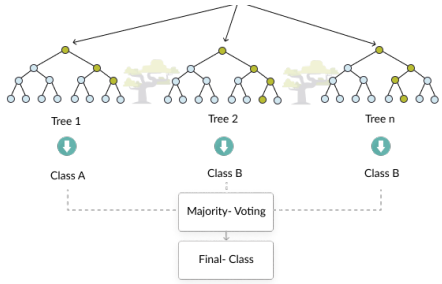
A more advanced version of the decision tree, which addresses overfitting by growing a large number of trees with random variations, then selecting and aggregating the best-performing decision trees. The “forest” is an ensemble of decision trees, typically done using a technique called “bagging”.

Strengths

Provides the strengths of the decision tree algorithm, and is very effective at preventing overfitting and thus much more accurate, even compared to a decision tree with extensive manual pruning.

Weaknesses

Not intuitive, difficult to understand why the model generates a specific outcome.



Naive Bayes Classifier

Classifier type

Multiclass

How it works

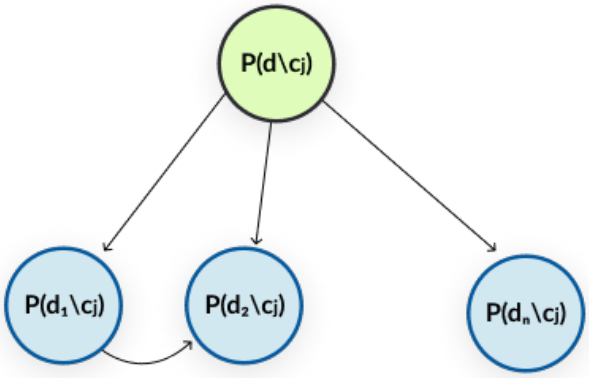
A probability-based classifier based on the Bayes algorithm. According to the concept of dependent probability, it calculates the probability that each of the features of a data point (the input variables) exists in each of the target classes. It then selects the category for which the probabilities are maximal. The model is based on an assumption (which is often not true) that the features are conditionally independent.

Strengths

Simple to implement and computationally light—the algorithm is linear and does not involve iterative calculations. Although its assumptions are not valid in most cases, Naive Bayes is surprisingly accurate for a large set of problems, scalable to very large data sets, and is used for many NLP models. Can also be used to construct multi-layer decision trees, with a Bayes classifier at every node.

Weaknesses

Very sensitive to the set of categories selected, which must be exhaustive. Problems where categories may be overlapping or there are unknown categories can dramatically reduce accuracy.



Classification with neural networks - Naive Bayes classifier

k-Nearest Neighbor (KNN)

Classifier type

Multiclass

How it works

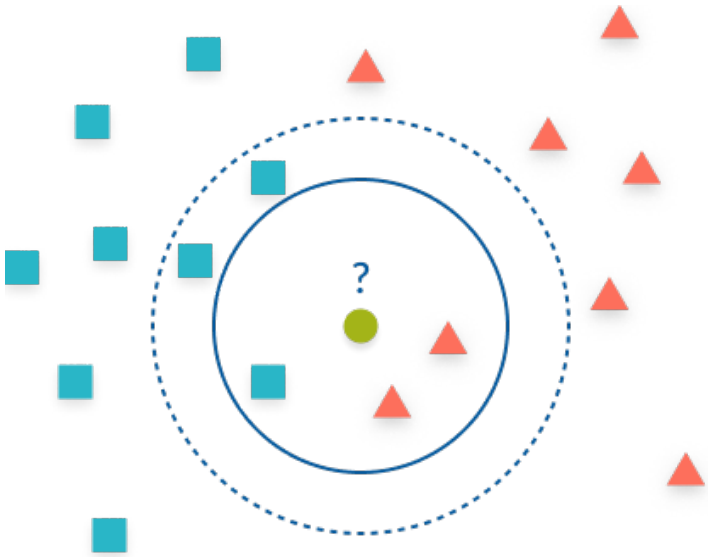
Classifies each data point by analyzing its nearest neighbors from the training set. The current data point is assigned the class most commonly found among its neighbors. The algorithm is non-parametric (makes no assumptions on the underlying data) and uses lazy learning (does not pre-train, all training data is used during classification).

Strengths

Very simple to implement and understand, and highly effective for many classification problems, especially with low dimensionality (small number of features or input variables).

Weaknesses

KNN's accuracy is not comparable to supervised learning methods. Not suitable for high dimensionality problems. Computationally intensive, especially with a large training set.



Classification with Neural Networks - k-Nearest Neighbor

Artificial Neural Networks and Deep Neural Networks

Classifier type

Either binary or multiclass

How it works

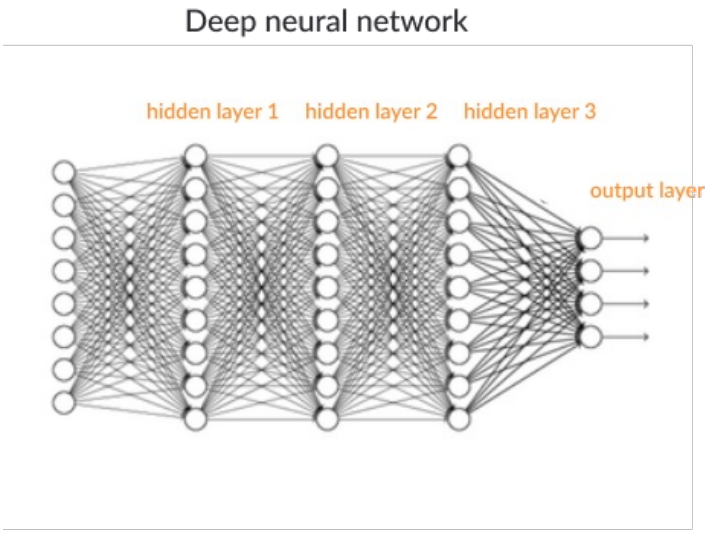
Artificial neural networks are built of simple elements called neurons, which take in a real value, multiply it by a weight, and run it through a non-linear activation function. By constructing multiple layers of neurons, each of which receives part of the input variables, and then passes on its results to the next layers, the network can learn very complex functions. Theoretically, a neural network is capable of learning the shape of just any function, given enough computational power.

Strengths

Very effective for high dimensionality problems, able to deal with complex relations between variables, non-exhaustive category sets and complex functions relating input to output variables. Powerful tuning options to prevent over- and under-fitting.

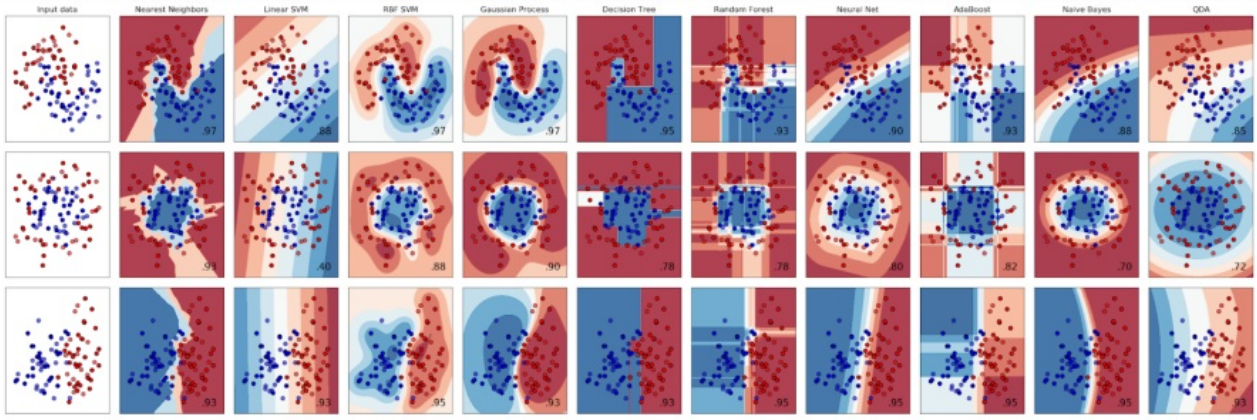
Weaknesses

Theoretically complex, difficult to implement (although deep learning frameworks are readily available that do the work for you). Non-intuitive and requires expertise to tune. In some cases requires a large training set to be effective.



Neural Networks and Regular Machine Learning Classifiers in the Real World

In real-world machine learning projects, you will find yourself iterating on the same classification problem, using different classifiers, and different parameters or structures of the same classifier. If you restrict yourself to "regular" classifiers besides neural networks, you can use great open source libraries like scikit-learn, which provide built-in implementations of all popular classifiers, and are relatively easy to get started with.



Scikit-Learn Library for Classification using Traditional Algorithms and Shallow Neural Networks

Visualizations of classifiers available in the scikit-learn package (source: scikit-learn) If you go down the neural network path, you will need to use the "heavier" deep learning frameworks such as Google's TensorFlow, Keras and PyTorch. These frameworks support both ordinary classifiers like Naive Bayes or KNN, and are able to set up neural networks of amazing complexity with only a few lines of code. While these frameworks are very powerful, each of them has operating concepts you'll need to learn, and each has its learning curve. There are additional challenges when running any machine learning project at scale:

Tracking progress across multiple experiments and storing source code, metrics and parameters. Machine learning experiments, especially neural networks, require constant trial and error to get the model right and it's easy to get lost as you create more and more experiments with multiple variations of each.

Running experiments across multiple machines—some classification algorithms, such as KNN and neural networks, are computationally intensive. As you scale up to real projects you'll have to run experiments on multiple machines. Managing those machines can be difficult.

Manage training data—if you're classifying images, video or large quantities of unstructured data, the training data itself can get big and storage and data transfer will become an issue.

Neural Network Bias: Bias Neuron, Overfitting and Underfitting

In this article you'll learn:

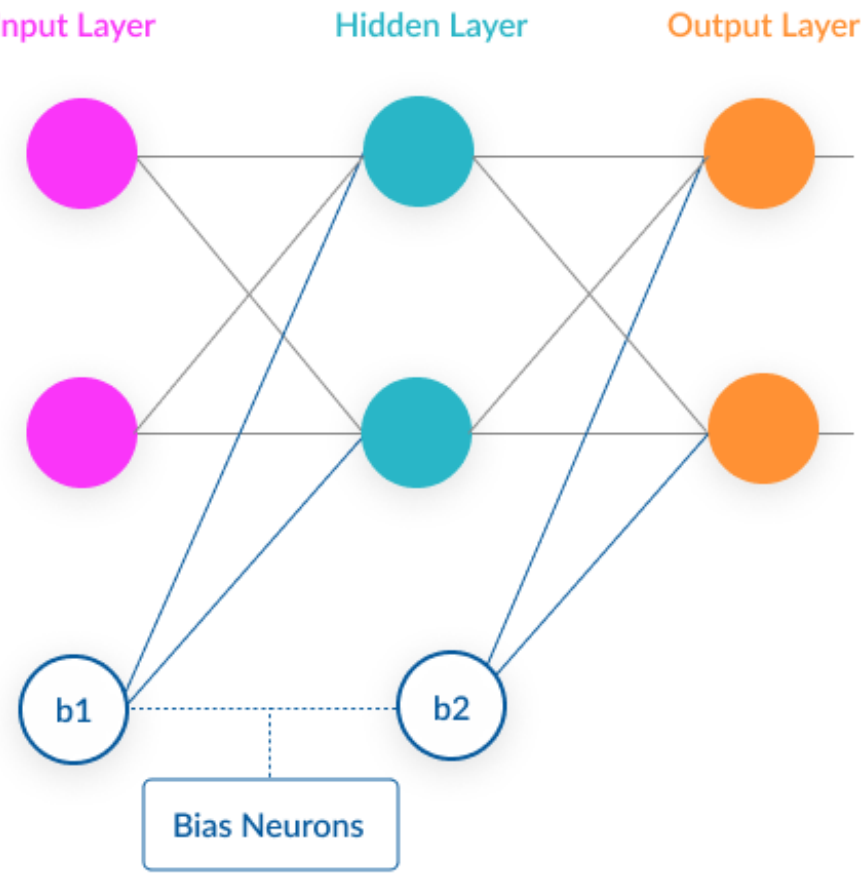
What Do We Mean by the Word Bias?

The word bias can have several meanings in neural networks. We'll explore the two most common meanings: bias as a bias neuron and bias as the bias vs. variance statistic.

Bias Neuron in a Neural Network

A neural network is a mathematical construct that can approximate almost any function, and generate predictions for complex problems.

A neural network is composed of neurons, which are very simple elements that take in a numeric input, apply an activation function to it, and pass it on to the next layer of neurons in the network. Below is a very simple neural network structure.



The bias neuron is a special neuron added to each layer in the neural network, which simply stores the value of 1. This makes it possible to move or "translate" the activation function left or right on the graph.

Without a bias neuron, each neuron takes the input and multiplies it by a weight, with nothing else added to the equation. So, for example, it is not possible to input a value of 0 and output 2. In many cases, it is necessary to move the entire activation function to the left or right to generate the required output values—this is made possible by the bias.

Although neural networks can work without bias neurons, in reality, they are almost always added, and their weights are estimated as part of the overall model.

To see a more complete example of a neural network and a bias neuron, see Creating a Neural Network Model with Bias Neuron below.

The Bias-Variance Tradeoff—Overfitting and Underfitting

Bias and variance are two statistical concepts that are important for all types of machine learning, including neural networks.

Basic Concepts: the Training and Validation Error

Before we introduce bias and variance, let's start with a few basic concepts:

Training Set

A training set is a group of sample inputs you can feed into the neural network in order to train the model. The neural network learns your inputs and finds weights for the neurons that can result in an accurate prediction.

Training Error

The error is the difference between the known correct output for the inputs and the actual output of the neural network. During the course of training, the training error is reduced until the model produces an accurate prediction for the training set.

Validation Set

A validation set is another group of sample inputs which were not included in training and preferably are different from the samples in the training set. This is a "real life exercise" for the model: Can it generate correct predictions for an unknown set of inputs?

Validation Error

The nice thing is that for the validation set, the correct outputs are already known. So it's

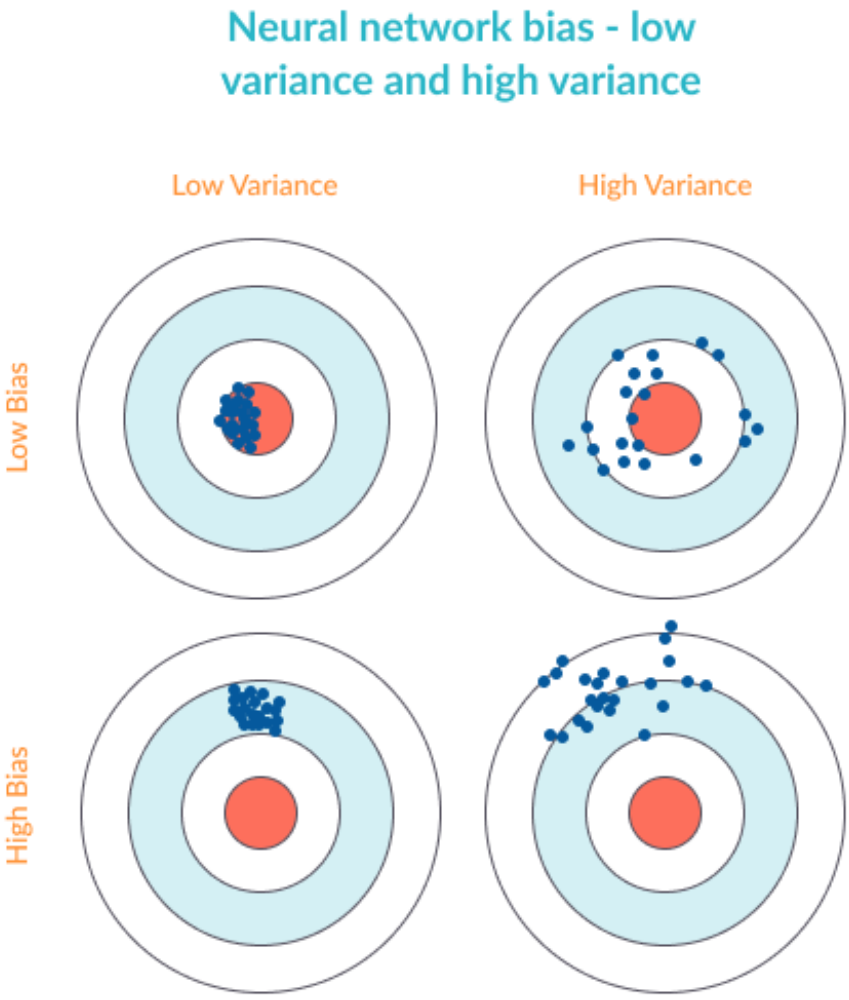
easy to compare the known correct prediction for the validation set with the actual model prediction—the difference between them is the validation error.

Practically, when training a neural network model, you will attempt to gather a training set that is as large as possible and resembles the real population as much as possible. You will then break up the training set into at least two groups: one group will be used as the training set and the other as the validation set.

Definition of Bias vs. Variance

Bias—high bias means the model is not "fitting" well on the training set. This means the training error will be large. Low bias means the model is fitting well, and training error will be low.

Variance—high variance means that the model is not able to make accurate predictions on the validation set. The validation error will be large. Low variance means the model is successful in breaking out of its training data.



Overfitting and Underfitting

To take a real-life example, borrowed from the excellent post by William Koehrsen, **overfitting** is like trying to learn the English language by using a very specific training set. For example, reading all the plays by Shakespeare. Someone who learns the language using only Shakespeare, as an example, will learn a very specialized form of English and may not be able to understand other English text.

To understand **underfitting**, consider someone who tries to learn English by listening to Seinfeld episodes, but ignoring most of the sentences and using only those that start with common words like “a”, “the”, “I”, etc. This person would have a very limited understanding even of those sentences themselves, not to mention a poor ability to understand English text in general.

Overfitting happens when the neural network is very good at learning its training set, but cannot generalize beyond the training set (known as the generalization problem).

The symptoms of overfitting are:

- **Low bias**
  accurate predictions for the training set
- **High variance**
  poor ability to generate predictions for the validation set

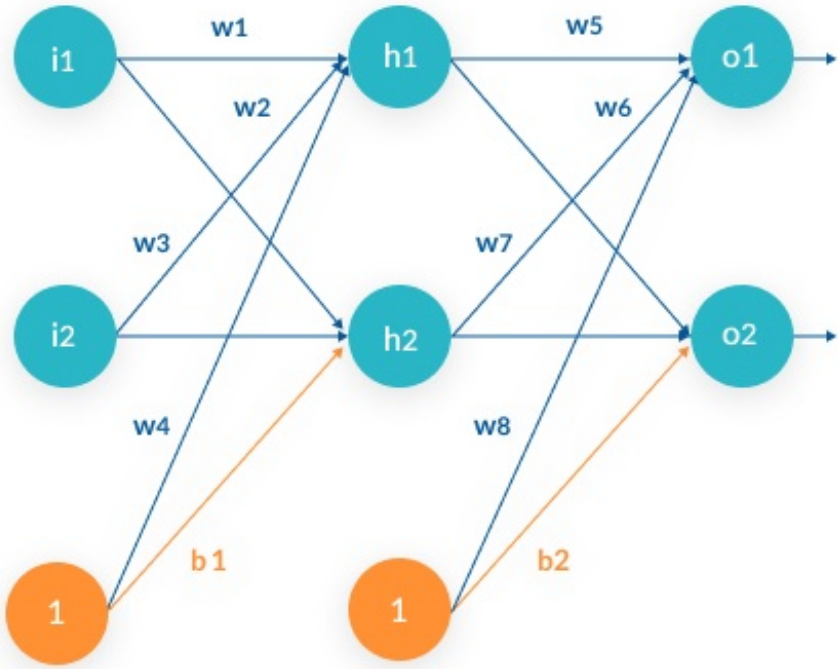
Underfitting happens when the network is not able to generate accurate predictions on the training set—not to mention the validation set.

The symptoms of underfitting are:

- **high bias**
  poor predictions for the training set
- **High variance**
  poor predictions for the validation set

To learn more about how to combat overfitting (by reducing variance) and underfitting (by reducing bias and variance), see Managing Bias and Variance below.

A Simple Neural Network Model with a Bias Neuron



The image above is a simple neural network that accepts two inputs which can be real values between 0 and 1 (in the example, 0.05 and 0.10), and has three neuron layers: an input layer (neurons i1 and i2), a hidden layer (neurons h1 and h2), and an output layer (neurons o1 and o2).

How do the neurons work? Each neuron is a very simple component which does nothing but executes the activation function. There are several commonly-used activation functions; for example, this is the sigmoid function:

$$f(x) = 1 / 1 + \exp(-x)$$

Go in-depth: Learn more in our guide on Activation Functions

When the inputs are fed into the neuron, they are multiplied by **weights**. For example, the weight for neuron i1 feeding into h1 is 0.15. So the input of 0.05 will be multiplied by 0.15 giving 0.333, and then fed into the activation function of neuron h1. The weights are learned in an iterative process called backpropagation.

Go in-depth: Learn more in our guide on Backpropagation

Where are the bias neurons? As explained above, biases in neural networks are extra neurons added to each layer, which store the value of 1. In our example, the bias neurons are b1 and b2 at the bottom. They also have weights attached to them (which are learned during backpropagation).

The forward pass

Our simple neural network works by:

1. Taking each of the two inputs

2. Multiplying by the first-layer weights—w1,2,3,4

3. Adding the bias value of 1 multiplied by the weight of bias neuron b1

4. Applying the activation function for neurons h1 and h2

5. Taking the output of h1 and h2, multiplying by the second layer weights—w5,6,7,8

6. Adding the bias value of 1 multiplied by the weight of bias neuron b2

7. The result of the calculation for each of the neurons o1 and o2 is the output or prediction of the neural network.

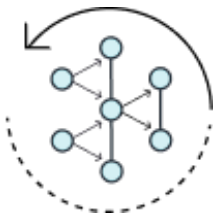
Managing Bias and Variance in Neural Networks

Now that we understand what bias and variance are and how they affect a neural network (if you skipped it, see our introduction to the Bias-Variance Tradeoff above), let's see a few practical ways to reduce bias and variance, and thus combat an overfitting or underfitting problem.

Methods to Avoid Overfitting in Neural Networks

The following are common methods used to improve the generalization of a neural network, so that it provides better predictive performance when applied to the unknown validation samples. The goal is to improve variance.

Retraining Neural Networks



Running the same neural network model on the same training set, but each time with different initial weights. The neural network with the lowest performance is likely to generalize the best when applied to the validation set.

Multiple Neural Networks

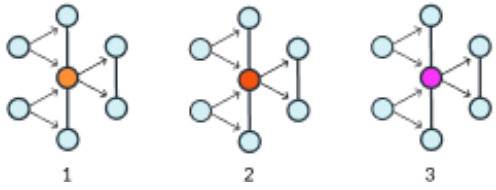
To avoid "quirks" created by a specific neural network model, you can train several neural networks in parallel, with the same structure but each with different initial weights, and average their outputs. This can be especially helpful for small, noisy datasets.

Early Stopping

Divide training data into a training set and a validation set and start training the model. Monitor the error on the validation set after each training iteration. Normally, the validation error and training set errors decrease during training, but when the network begins to overfit the data, the error on the validation set begins to rise. Early stopping occurs in one of two cases:

- If the training is unsuccessful, for example in a case where the error rate increases gradually over several iterations.
- If the training's improvement is insignificant, for example, the improvement rate is lower than a set threshold.

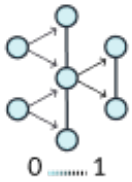
Regularization



Regularization is a slightly more complex technique which involves modifying the error function (usually calculated as the sum of squares on the errors for the individual training or validation samples).

Without getting into the math, the trick is to add a term to the error function, which is intended to decrease the weights and biases, smoothing outputs and making the network less likely to overfit.

Tuning Performance Ratio



The regularization term includes a performance ratio γ—this is a parameter that defines by how much the network will be smoothed out.

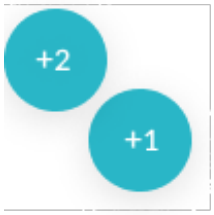
- You can manually set a performance ratio—if you set it to zero, the network won't be regularized at all, and if you set it to 1 it will not fit the training data at all.
- It is also possible to automatically learn the optimal value of the performance ratio and then apply it to the network to achieve a balance.

Methods to Avoid Underfitting in Neural Networks—Adding Parameters, Reducing Regularization Parameter

What happens when the neural network is "not working"—not managing to predict even its training results? This is known as underfitting and reflects a **low bias** and **low variance** of the model. The following are common methods for improving fit—the goal is to increase bias and variance of the model.

Adding neuron layers or input parameters

For complex problems, adding neuron layers can help generate more complex predictions and improve the fit of the model. Another option, if relevant for your problem, is to increase the number of input parameters flowing into the model (for example, if you are processing an image, input all the pixels from a higher resolution image).



Adding more training samples, or improving their quality

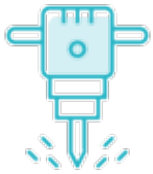
The more training samples you provide, and the better they represent the variance of parameters in the population at large, the better the network will learn.

Dropout

Another way to improve model fit is to randomly "kill" a certain percentage of the neurons in every training iteration. This can help improve generalization by ensuring that some of the information learned from the training set is randomly removed.

Decreasing regularization parameter

If you attempted to improve the model fit with regularization, you may have overdone it! Decrease the performance ratio to improve the fit and reduce bias.



Neural Network Bias in the Real World

When training a real neural network model, you will probably use a deep learning framework such as **Keras**, **Tensorflow**, **Caffe** or **Pytorch**. Here is how these frameworks typically handle bias neurons, overfitting and underfitting:

- **Bias neurons** are automatically added to models in most deep learning libraries, and trained automatically.
- **If you experience overfitting**, deep learning frameworks allow you to automatically execute the techniques we discussed—multiple neural networks, early stopping, regularization, and many more.
- **If you experience underfitting**, you can modify hyperparameters of the model, such as the number of neuron layers (network depth), dropout, regularization, and activation function.

Today's deep learning frameworks let you train models quickly and efficiently with only a few lines of code and tune their parameters conveniently by configuration. However, in real-world projects you are likely to run into some challenges:

Tracking experiment progress—training multiple models with different hyperparameters, you will find yourself managing many different configuration files and experiment variations, which can quickly get out of hand.

Running experiments across multiple machines—getting a model right requires a lot of trial and error. You will need to run models many times, and if training sets are large this requires major computing resources. Distributing an experiment across multiple machines, copying the relevant training and validation sets into each machine, and repeating, is a complex exercise.

Manage training data—storing groups of training samples, which can weigh in the Petabytes for some projects, dividing them into training and validation sets (which becomes more complex when you use cross-validation), and avoiding duplication and human error, is a challenge of its own.

Hyperparameters: Optimization Methods and Real World Model Management

Neural network hyperparameters shape how the network functions, and determine its accuracy and validity. Hyperparameters are an unsolved problem – there are various ways to optimize them, from manual trial and error to sophisticated algorithmic methods, and no industry consensus on what works best.

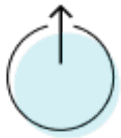
In this article you'll learn:

What is the Difference Between a Model Parameter and a Hyperparameter?

In neural networks, parameters are used to train the model and make predictions. There are two types of parameters:

Model parameters are internal to the neural network – for example, neuron weights. They are estimated or learned automatically from training samples. These parameters are also used to make predictions in a production model.

Hyperparameters are external parameters set by the operator of the neural network – for example, selecting which activation function to use or the batch size used in training. Hyperparameters have a huge impact on the accuracy of a neural network, there may be different optimal values for different values, and it is non-trivial to discover those values.



The simplest way to select hyperparameters for a neural network model is "manual search" – in other words, trial and error. New methods are evolving which use algorithms and optimization methods to discover the best hyperparameters. To learn more about these methods see Hyperparameter Tuning below.

List of Common Hyperparameters

This list assumes a basic knowledge of neural network concepts. For a refresh, see our in-depth neural network guide.

Hyperparameters related to neural network structure

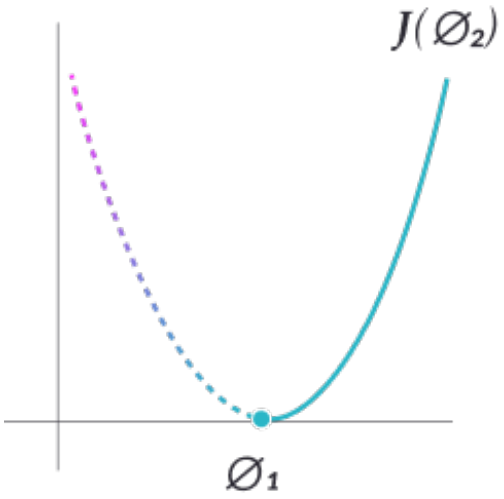
1. **Number of hidden layers**adding more hidden layers of neurons generally improves accuracy, to a certain limit which can differ depending on the problem.
2. **Dropout**what percentage of neurons should be randomly "killed" during each epoch to prevent overfitting.
3. **Neural network activation function** - which function should be used to process

the inputs flowing into each neuron. The activation function can impact the network's ability to converge and learn for different ranges of input values, and also its training speed.

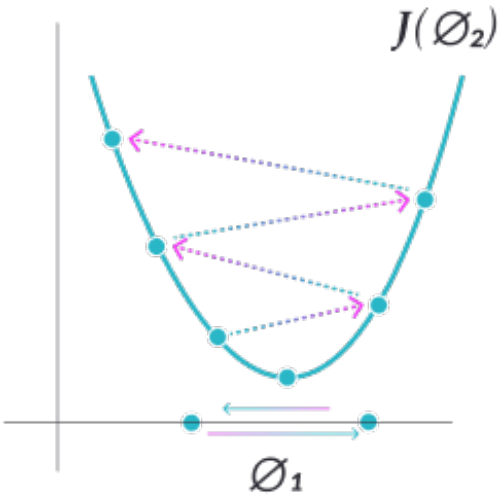
4. **Weights initialization** - it is necessary to set initial weights for the first forward pass. Two basic options are to set weights to zero or to randomize them. However, this can result in a vanishing or exploding gradient, which will make it difficult to train the model. To mitigate this problem, you can use a heuristic (a formula tied to the number of neuron layers) to determine the weights. A common heuristic used for the Tanh activation is called Xavier initialization.

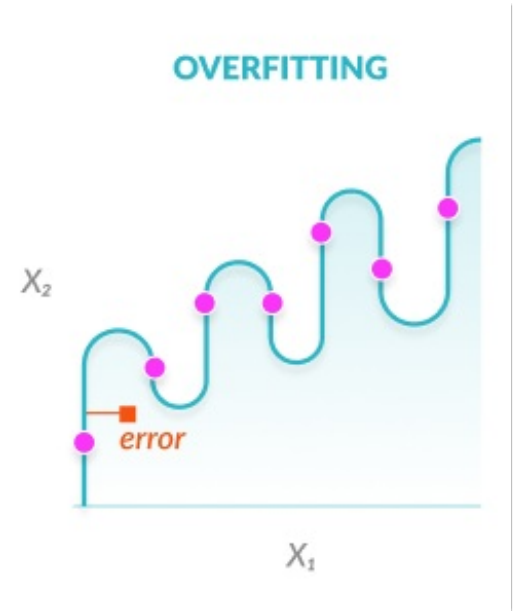
Hyperparameters related to training algorithm

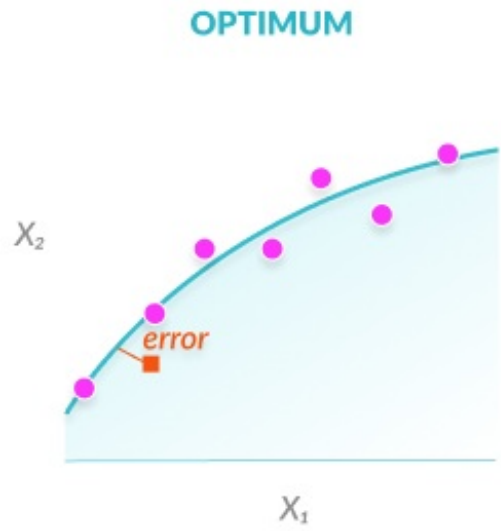
5. **Neural network learning rate** - how fast the backpropagation algorithm performs gradient descent. A lower learning rate makes the network train faster but might result in missing the minimum of the loss function.
6. **Deep learning epoch, iterations and batch size** - these parameters determine the rate at which samples are fed to the model for training. An epoch is a group of samples which are passed through the model together (forward pass) and then run through backpropagation (backward pass) to determine their optimal weights. If the epoch cannot be run all together due the size of the sample or complexity of the network, it is split into batches, and the epoch is run in two or more iterations. The number of epochs and batches per epoch can significantly affect model fit, as shown below.

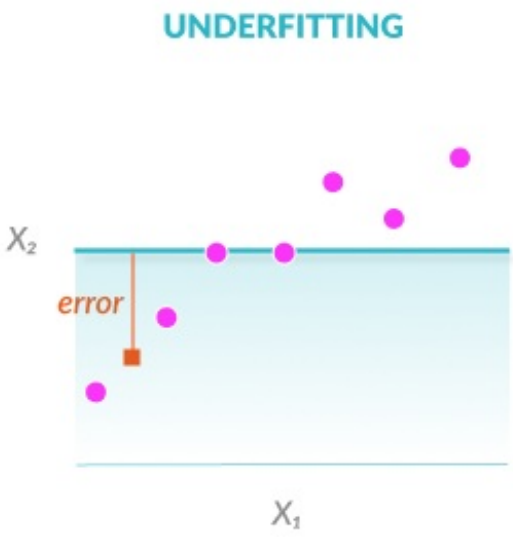


7. **Optimizer algorithm and neural network momentum**- when a neural network trains, it uses an algorithm to determine the optimal weights for the model, called an optimizer. The basic option is Stochastic Gradient Descent, but there are other options. Another common algorithm is Momentum, which works by waiting after a weight is updated, and updating it a second time using a delta amount. This speeds up training gradually, with a reduced risk of oscillation. Other algorithms are Nesterov Accelerated Gradient, AdaDelta and Adam.




OVERFITTING
X_2
error
X_1


OPTIMUM
X_2
error
X_1


UNDERFITTING
X_2
error
X_1

4 Methods of Hyperparameter Tuning in a Deep Neural Network

Tuning or optimizing hyperparameters involves finding the values of each hyperparameter which will help the model provide the most accurate predictions.

Optimization Metric and Validation

Hyperparameter tuning is always performed against an **optimization metric** or score. This is the metric you are trying to optimize when you try different hyperparameter values. Typically, the optimization metric is accuracy. However, if you blindly optimize for accuracy and ignore overfitting or underfitting, you'll get a highly accurate model (when applied to the training set) but which does not perform well with unknown samples. **Validation** helps ensure you are not optimizing for accuracy at the expense of model fit. To perform validation, the training samples are split into at least two parts: a training set and a validation set. The model is trained on the samples and then run on the validation set for testing. This allows you to gauge if the model is underfitting or overfitting. If the number of samples is small, you can use **cross validation** – this involves dividing the training set into multiple groups, for example 10 groups. You can then train the model on each of the 10 groups, and validate it against the other 9. By doing this for all 10 combinations, you can simulate a much larger training and validation set.

1. Manual Hyperparameter Tuning

Traditionally, hyperparameters were tuned manually by trial and error. This is still commonly done, and experienced operators can "guess" parameter values that will achieve very high accuracy for deep learning models. However, there is a constant search for better, faster and more automatic methods to optimize hyperparameters. **Pros:** Very simple and effective with skilled operators **Cons:** Not scientific, unknown if you have fully optimized hyperparameters

2. Grid Search

Grid search is slightly more sophisticated than manual tuning. It involves systematically testing multiple values of each hyperparameter, by automatically retraining the model for each value of the parameter. For example, you can perform a grid search for the optimal batch size by automatically training the model for batch sizes between 10-100 samples, in steps of 20. The model will run 5 times and the batch size selected will be the one which yields highest accuracy. **Pros:** Maps out the problem space and provides more opportunity for optimization **Cons:** Can be slow to run for large numbers of hyperparameter values

3. Random Search

According to a 2012 research study by James Bergstra and Yoshua Bengio, testing randomized values of hyperparameters is actually more effective than manual search or grid search. In other words, instead of testing systematically to cover "promising areas" of the problem space, it is preferable to test random values drawn from the entire problem space. **Pros:** According to the study, provides higher accuracy with less training cycles, for problems with high dimensionality **Cons:** Results are unintuitive, difficult to understand "why" hyperparameter values were chosen

4. Bayesian Optimization

Bayesian optimization (described by Shahriari, et al) is a technique which tries to approximate the trained model with different possible hyperparameter values. To simplify, bayesian optimization trains the model with different hyperparameter values, and observes the function generated for the model by each set of parameter values. It does this over and over again, each time selecting hyperparameter values that are slightly different and can help plot the next relevant segment of the problem space. Similar to sampling methods in statistics, the algorithm ends up with a list of possible hyperparameter value sets and model functions, from which it predicts the optimal function across the entire problem set. **Pros:** The original study and practical experience from the industry shows that bayesian optimization results in significantly higher accuracy compared to random search. **Cons:** Like random search, results are not intuitive and difficult to improve on, even by trained operators

Hyperparameter Optimization in the Real World

In a real neural network project, you will have three practical options:

1. Performing manual optimization
2. Leveraging hyperparameter optimization techniques in the deep learning framework of your choice. The framework will report on hyperparameter values discovered, their accuracy and validation scores
3. Using third party hyperparameter optimization tools

> **If you use Keras**, the following libraries provide different options for hyperparameter optimization: Hyperopt, Kopt and Talos

For third party optimization tools, see this post by Mikko Kotila.

If you use Tensorflow, you can leverage open source libraries such as GPflowOpt which provides bayesian optimization, and commercial solutions like Google's Cloud Machine Learning Engine.

Managing Models, Experiments and Data for Hyperparameter Optimization

Whether you tune hyperparameters manually or using an automated tool, you will need to run experiments repeatedly to test the results, constantly shifting data between training and validation sets. This raises significant challenges:

Parallelization for a large numbers of experiments – to perform grid search or bayesian optimization for a realistic model with a large number of training samples, you will need to parallelize across multiple machines, either on-premise or in the cloud. Provisioning and managing these machines quickly becomes a burden.

Manage training data – training data needs to be dynamically copied to the specific machines that run each experiment. For large scale hyperparameter optimization this can be very difficult to manage, especially if the training samples are images or videos which can reach petabyte scale.

Generative Adversarial Networks

missinglink.ai/guides/neural-network-concepts/generative-adversarial-networks

The world around us is full of detail, and we intuitively understand the difference between natural and fake objects. The sheer complexity of an individual voice or face makes it almost unthinkable to recreate an accurate representation. So how do you create an image, voice, or text from scratch, without anyone noticing that it is artificial? You can do it by using two neural networks pitted against each other, challenging each other to understand what makes objects appear realistic.

In this article:

What Is a GAN?

A Generative Adversarial Network (GAN) is a deep learning (DL) architecture comprising two competing neural networks within the framework of a zero-sum game. It is a form of unsupervised learning first introduced by Ian J. Goodfellow and his colleagues in 2014. Since then, this technology has seen rapid advances.

Generative adversarial networks, like other generative models, can artificially generate artifacts, such as images, video, and audio, which resemble human-generated artifacts. The objective is to produce a complex output from a simple input, with the highest possible level of accuracy. For example, translating a few random numbers into a realistic image of a face.

GAN works by training neural networks against each other. One network creates a fake artifact, and the other network attempts to distinguish the fake object from the real object. At first, there is a clear difference between the real object and the fake, but as the dual network trains itself over time, it learns to create more realistic artifacts, until the fake artifacts manage to “fool” the system and are taken to be real.

Examples of GAN Applications

While generative adversarial networks have fewer business-critical applications when compared to other deep learning models, you can use GAN to create or enhance objects for a variety of artistic applications. For example, you can convert black-and-white images to color and increase their resolution, or train a bot to author a blog post.

Notable applications of GAN include:

- **Data augmentation**—you can train a GAN to generate new sample images from your data, augmenting your dataset. Once your GAN is mature, you can use the images it generates to help train other networks.
- **Text-to-image generation**—uses include producing films or comics by automatically generating a sequence of images based on a text input.
- **Generating faces**—NVIDIA researchers trained a GAN using over 200,000 sample images of celebrity faces, which was then able to generate photorealistic images of people who have never actually existed.
- **Image-to-image translation**—the GAN learns to map a pattern from input images onto output images. For example, you can use it to convert an image into the style of a famous painter like Van Gogh or Monet or to transform a horse into a zebra.
- **E-Commerce and industrial design**—you can use the GAN for suggesting merchandise and creating new 3D products based on product data. For example, you can generate new styles of clothes to help meet consumer demand.

How Do GANs Work?

The two neural networks that make up the GAN are called the generator and the discriminator. The GAN generator creates new data instances and the discriminator evaluates their authenticity, or whether they belong in the dataset. The GAN discriminator is a fully connected neural network that classifies whether an image is real (1) or generated (0).

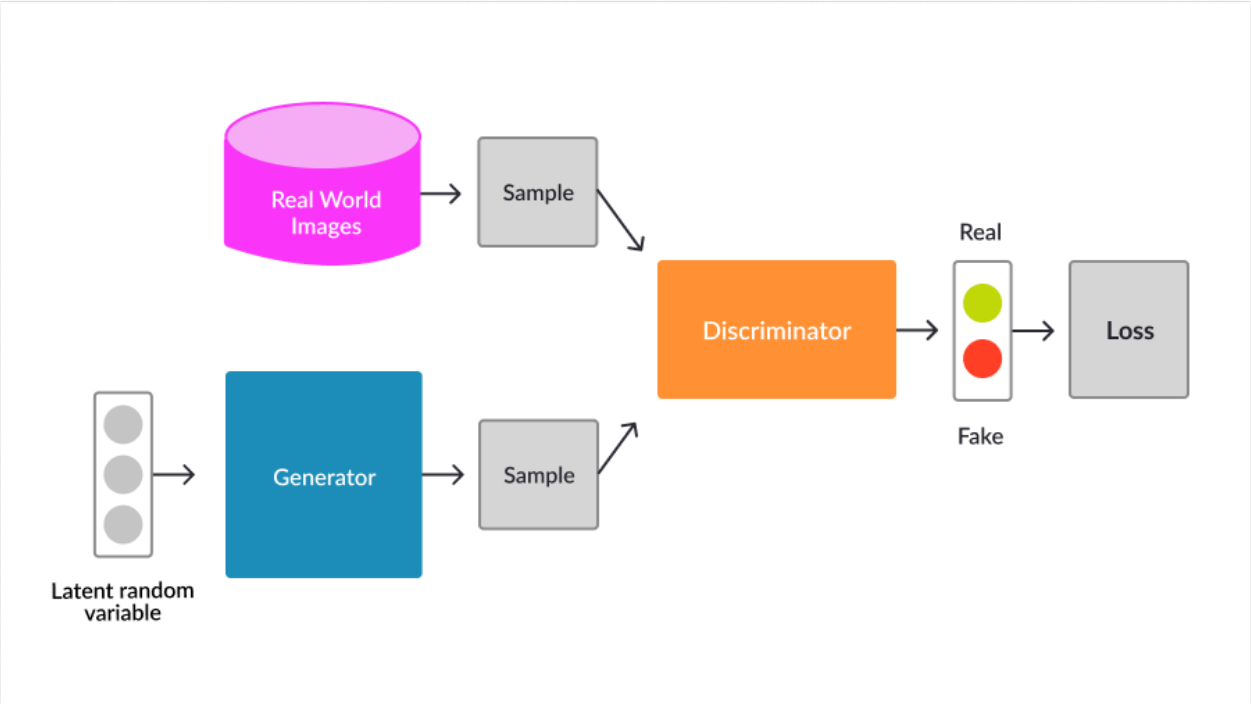
To establish a generative adversarial network, you first need to identify your desired end output and provide an initial training dataset. The GAN starts by learning on a simple distribution of points in two dimensions, and with training will eventually be able to mimic any distribution of data.

While your main training objective is to train the generator, both networks benefit from the training loop. In a process known as backpropagation, each network trains and reinforces the other until the discriminator can no longer distinguish between the real and the generated images.

In the first phase, you need to train the discriminator to identify probability distribution. You can use images from the MNIST database along with the generated images to train the discriminator. Once the discriminator has achieved a basic level of maturity, you can start training the generator against the discriminator. The generator learns to map a random data distribution from a latent sample (a series of randomly generated numbers). You can also set the discriminator to non-trainable while training the generator.

For a typical example of a GAN process, you can feed a D-dimensional noise vector taken from the latent sample into the generator, which then converts it into an image. The discriminator then processes and classifies it, and the feedback helps the generator

improve its capabilities. A crucial advantage of GAN is its randomness, which allows it to generate an entirely new object, rather than simply creating an exact replica of the input.



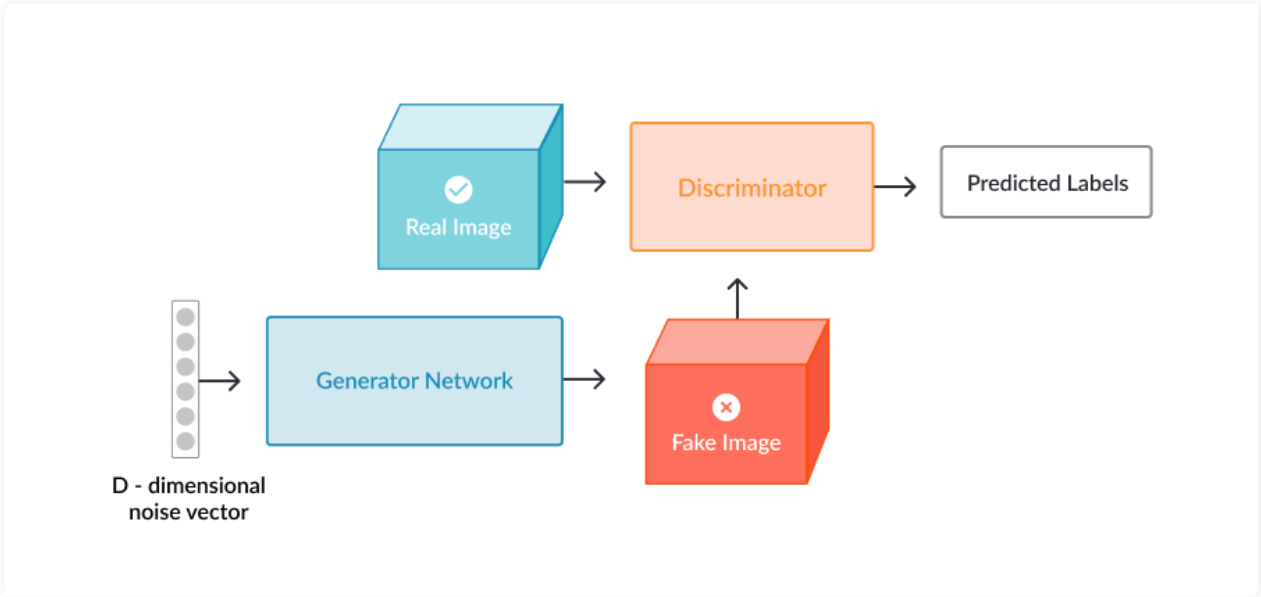
Popular Types of GANs

Since generative adversarial networks were first proposed, new designs have emerged that build on the basic GAN architecture. These new architectures improve on the accuracy of object generation or address specific design needs.

DCGAN

Convolutional neural networks (CNNs) are typically used for supervised computer vision tasks. Deep convolutional generative adversarial networks (DCGANs), which were introduced in a 2015 paper by Alec Radford, Luke Metz, and Soumith Chintala, incorporate convolutional networks into an unsupervised deep learning framework, and it is the first instance in which CNNs have achieved impressive results when used within a GAN.

This model involves major architectural changes, including the replacement of pooling layers with strided convolutions and fractional-strided convolutions; the use of batch normalization for both networks; and the removal of fully connected hidden layers. These changes stabilize training, allow for deeper generative models, and prevent mode collapse and internal covariate shift.



StyleGan

This model, based on a style mixing algorithm, enables a better understanding of generated outputs and produces more authentic-looking images than previous models. It also offers more control over specific features. The style-based generator generates the images gradually, in progressive layers, starting with low-resolution outputs and building up to high-resolution. For example, the coarse level may determine pose and general outline, the middle level may determine facial expressions and hairstyle, and the fine level may determine microfeatures.

The architecture includes a mapping network made up of eight fully connected layers, with an output equal in size to the input layer. The mapping network encodes the input vector into an intermediate vector that can control various features. An adaptive instance normalization (AdaIN) style module converts the encoded information from the mapping network into a generated image.

BigGan:

The latest development in GAN image generation, this model is distinguished by the sheer computing power that supports it. It produces images with unprecedented high fidelity and a low variety gap. To illustrate this point, BigGAN has achieved an Inception Score of 166.3, compared with the previous record of 52.52.

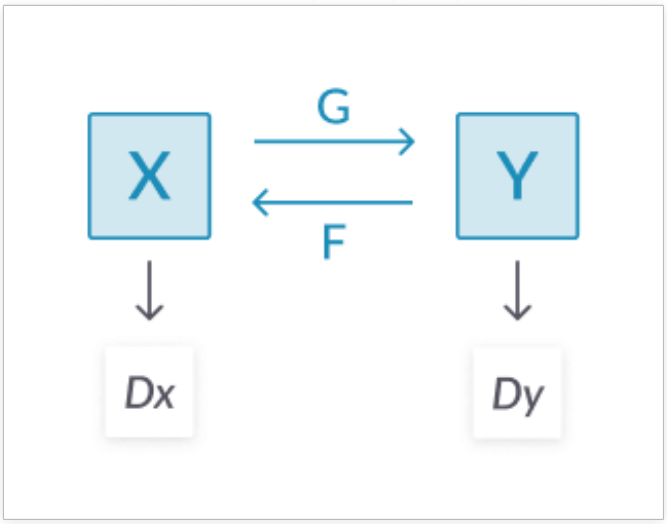
The success of BigGAN boils down to computational factors, rather than algorithmic ones. By adding more nodes to the neural network, you can increase complexity while training the system to pick up subtle differences in texture, allowing it to recreate realistic skin, hair, or water. However, this type of architecture is not yet viable for common use. In his experiment, Ph.D. student Andrew Brock used Google's Tensor Processing Units (TPUs), which consume large amounts of energy.

CycleGAN:

This was first proposed by Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. You can apply cycle generative adversarial networks (CycleGAN) to various image-to-image translation use-cases. For example, you can convert a photograph into a painting, or a painting into a photograph.

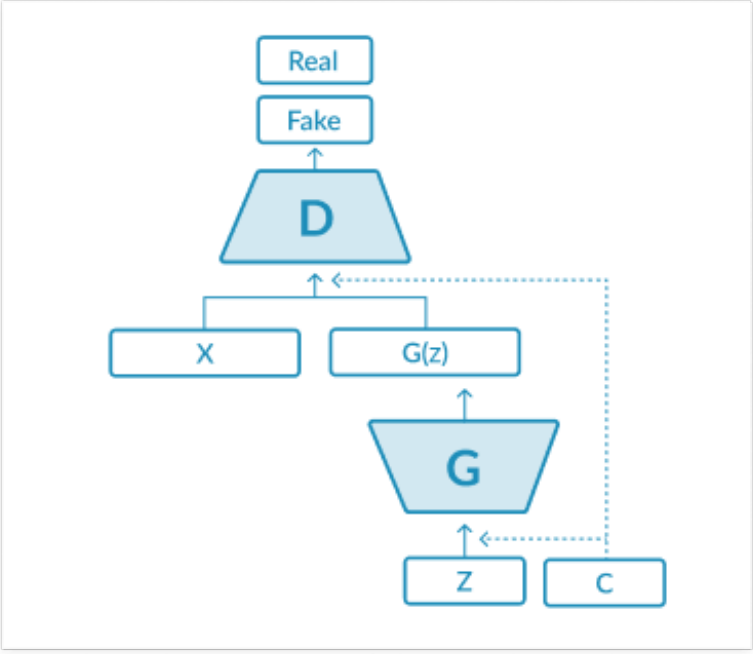
The architecture consists of two adversarial mappings: G : X -> Y and F : Y -> X. G generates outputs from X, and Domain Y evaluates if they are real or fake. F generates outputs from Y, and Domain X checks if they are real or fake. The GAN tries to minimize the cyclic loss incurred when translating an input from X to Y and back again.

Conditional GAN:



A conditional generative adversarial network (CGAN), introduced by Mehdi Mirza and Simon Osindero, involves applying conditional settings to the GAN. Both the generator and discriminator are conditioned on auxiliary information, giving the user greater control over the output.

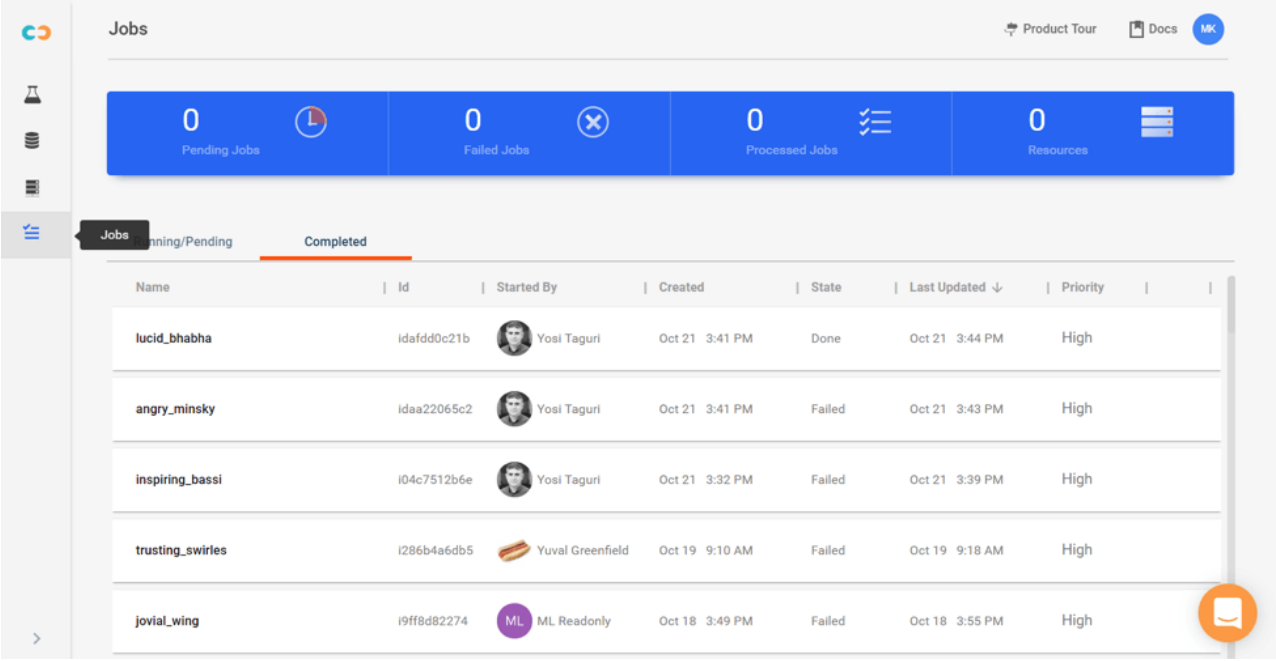
In the CGAN architecture, the noise vector input is embedded with a class label, typically a one-hot encoded vector. The generator has an added Y parameter for generating the corresponding data, and the input is labeled to help the discriminator distinguish between the real and fake data. This allows the model to learn multimodal mapping of outputs from inputs with reference to contextual information. For example, conditioning MNIST images on class labels.



Scaling Up GANs with MissingLink

Large, computationally intensive neural networks like GAN can be tricky to manage, and it often takes hours or even days to train them. Setting up your deep learning model on-premise requires manually running experiments and tracking your resource utilization, while cloud-based machines require constant management to ensure that you don't waste time and money.

MissingLink offers a solution for GAN and other deep learning models. It is a deep learning platform that can run generative adversarial networks on multiple machines and lets you scale your network.



You can set up jobs in the MissingLink dashboard, select the on-premise or cloud machines you wish to use, and the jobs will run automatically. You can train your GAN model in minutes while MissingLink runs experiments continuously to avoid idle time.

Image Segmentation in Deep Learning: Methods and Applications

missinglink.ai/guides/neural-network-concepts/image-segmentation-deep-learning-methods-applications

Modern Computer Vision technology, based on AI and deep learning methods, has evolved dramatically in the past decade. Today it is used for applications like image classification, face recognition, identifying objects in images, video analysis and classification, and image processing in robots and autonomous vehicles. Many computer vision tasks require intelligent segmentation of an image, to understand what is in the image and enable easier analysis of each part. Today's image segmentation techniques use deep learning to understand, at a level unimaginable only a decade ago, exactly which real-world object is represented by each pixel of an image. Deep learning can learn patterns in visual inputs in order to predict object classes that make up an image. The main deep learning architecture used for image processing is a Convolutional Neural Network (CNN), or specific CNN frameworks like AlexNet, VGG, Inception, and ResNet. Deep learning computer vision models are typically trained and executed on specialized graphics processing units (GPUs) to reduce computation time. **In this article you will learn about:**

- **What is image segmentation?**
- **Old-school image segmentation methods**
- **Image segmentation methods in deep learning**
- **Image segmentation applications**

What is Image Segmentation?

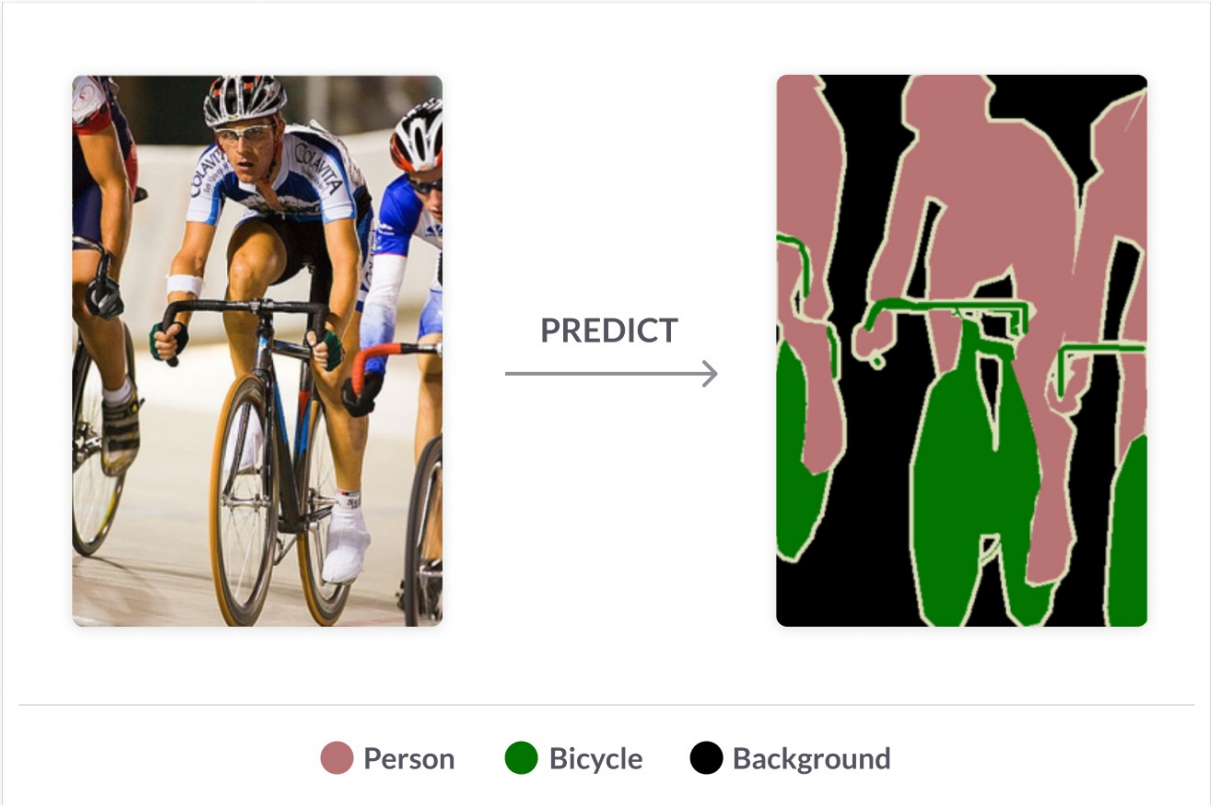
Image segmentation is a critical process in computer vision. It involves dividing a visual input into segments to simplify image analysis. Segments represent objects or parts of objects, and comprise sets of pixels, or "super-pixels". Image segmentation sorts pixels into larger components, eliminating the need to consider individual pixels as units of observation. There are three levels of image analysis:

- **Classification** – categorizing the entire image into a class such as "people", "animals", "outdoors"
- **Object detection** – detecting objects within an image and drawing a rectangle around them, for example, a person or a sheep.
- **Segmentation** – identifying parts of the image and understanding what object they belong to. Segmentation lays the basis for performing object detection and classification.

Semantic Segmentation vs. Instance Segmentation

Within the segmentation process itself, there are two levels of granularity:

> **Semantic segmentation**—classifies all the pixels of an image into meaningful classes of objects. These classes are "semantically interpretable" and correspond to real-world categories. For instance, you could isolate all the pixels associated with a cat and color them green. This is also known as dense prediction because it predicts the meaning of each pixel.



> **Instance segmentation**—identifies each instance of each object in an image. It differs from semantic segmentation in that it doesn't categorize every pixel. If there are three cars in an image, semantic segmentation classifies all the cars as one

instance, while instance segmentation identifies each individual car.

Old-School Image Segmentation Methods

There are additional image segmentation techniques that were commonly used in the past but are less efficient than their deep learning counterparts because they use rigid algorithms and require human intervention and expertise. These include:

- **Thresholding**—divides an image into a foreground and background. A specified threshold value separates pixels into one of two levels to isolate objects. Thresholding converts grayscale images into binary images or distinguishes the lighter and darker pixels of a color image.
- **K-means clustering**—an algorithm identifies groups in the data, with the variable K representing the number of groups. The algorithm assigns each data point (or pixel) to one of the groups based on feature similarity. Rather than analyzing predefined groups, clustering works iteratively to organically form groups.
- **Histogram-based image segmentation**—uses a histogram to group pixels based on "gray levels". Simple images consist of an object and a background. The background is usually one gray level and is the larger entity. Thus, a large peak represents the background gray level in the histogram. A smaller peak represents the object, which is another gray level.
- **Edge detection**—identifies sharp changes or discontinuities in brightness. Edge detection usually involves arranging points of discontinuity into curved line segments, or edges. For example, the border between a block of red and a block of blue.

How Deep Learning Powers Image Segmentation Methods

Modern image segmentation techniques are powered by deep learning technology. Here are several deep learning architectures used for segmentation: **Convolutional Neural Networks (CNNs)** Image segmentation with CNN involves feeding segments of an image as input to a convolutional neural network, which labels the pixels. The CNN cannot process the whole image at once. It scans the image, looking at a small "filter" of several pixels each time until it has mapped the entire image. To learn more see our in-depth guide about Convolutional Neural Networks. **Fully Convolutional Networks (FCNs)** Traditional CNNs have fully-connected layers, which can't manage different input sizes. FCNs use convolutional layers to process varying input sizes and can work faster. The final output layer has a large receptive field and corresponds to the height and width of the image, while the number of channels corresponds to the number of classes. The convolutional layers classify every pixel to determine the context of the image, including the location of objects. **Ensemble learning** Synthesizes the results of two or more related analytical models into a single spread. Ensemble learning can improve prediction accuracy and reduce generalization error. This enables accurate classification and segmentation of images. Segmentation via ensemble learning attempts to generate a set of weak base-learners which classify parts of the image, and combine their output,

instead of trying to create one single optimal learner. **DeepLab** One main motivation for DeepLab is to perform image segmentation while helping control signal decimation—reducing the number of samples and the amount of data that the network must process. Another motivation is to enable multi-scale contextual feature learning—aggregating features from images at different scales. DeepLab uses an ImageNet pre-trained residual neural network (ResNet) for feature extraction. DeepLab uses atrous (dilated) convolutions instead of regular convolutions. The varying dilation rates of each convolution enable the ResNet block to capture multi-scale contextual information. DeepLab is comprised of three components:

- **Atrous convolutions**—with a factor that expands or contracts the convolutional filter's field of view.
- **ResNet**—a deep convolutional network (DCNN) from Microsoft. It provides a framework that enables training thousands of layers while maintaining performance. The powerful representational ability of ResNet boosts computer vision applications like object detection and face recognition.
- **Atrous spatial pyramid pooling (ASPP)**—provides multi-scale information. It uses a set of atrous convolutions with varying dilation rates to capture long-range context. ASPP also uses global average pooling (GAP) to incorporate image-level features and add global context information.

SegNet neural network An architecture based on deep encoders and decoders, also known as semantic pixel-wise segmentation. It involves encoding the input image into low dimensions and then recovering it with orientation invariance capabilities in the decoder. This generates a segmented image at the decoder end.

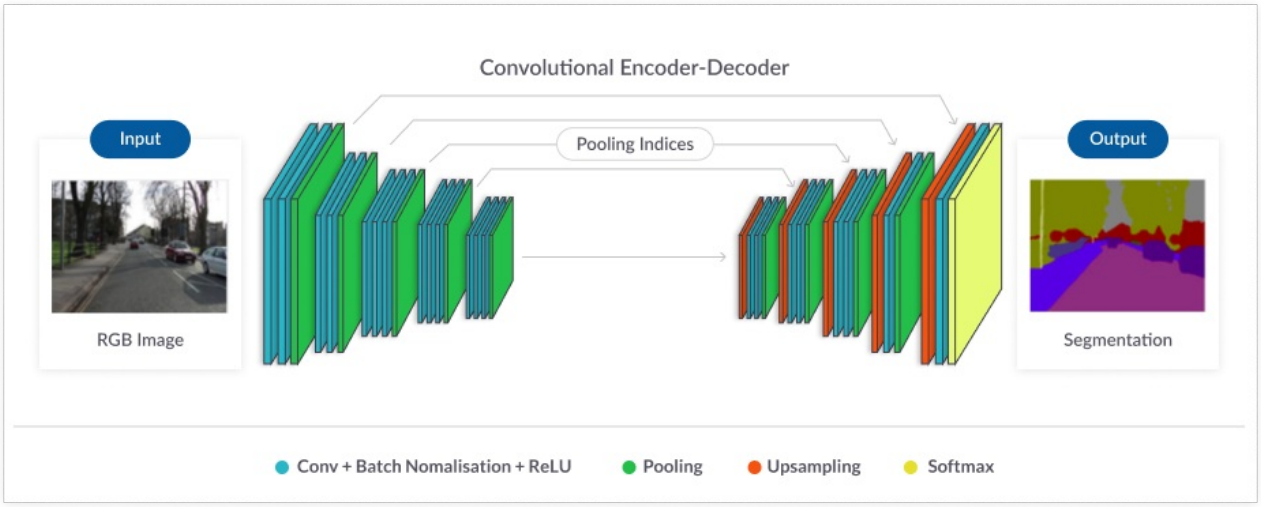


Image Segmentation Applications

Image segmentation helps determine the relations between objects, as well as the context of objects in an image. Applications include face recognition, number plate identification, and satellite image analysis. Industries like retail and fashion use image segmentation, for example, in image-based searches. Autonomous vehicles use it to understand their surroundings.

Object Detection and Face Detection

These applications involve identifying object instances of a specific class in a digital image. Semantic objects can be classified into classes like human faces, cars, buildings, or cats.

- **Face detection**—a type of object-class detection with many applications, including biometrics and autofocus features in digital cameras. Algorithms detect and verify the presence of facial features. For example, eyes appear as valleys in a gray-level image.
- **Medical imaging**—extracts clinically relevant information from medical images. For example, radiologists may use machine learning to augment analysis, by segmenting an image into different organs, tissue types, or disease symptoms. This can reduce the time it takes to run diagnostic tests.
- **Machine vision**—applications that capture and process images to provide operational guidance to devices. This includes both industrial and non-industrial applications. Machine vision systems use digital sensors in specialized cameras that allow computer hardware and software to measure, process, and analyze images. For example, an inspection system photographs soda bottles and then analyzes the images according to pass-fail criteria to determine if the bottles are properly filled.

Video Surveillance—video tracking and moving object tracking

This involves locating a moving object in video footage. Uses include security and surveillance, traffic control, human-computer interaction, and video editing.

- **Self-driving vehicles**—autonomous cars must be able to perceive and understand their environment in order to drive safely. Relevant classes of objects include other vehicles, buildings, and pedestrians. Semantic segmentation enables self-driving cars to recognize which areas in an image are safe to drive.
- **Iris recognition**—a form of biometric identification that recognizes the complex patterns of an iris. It uses automated pattern recognition to analyze video images of a person's eye.
- **Face recognition**—identifies an individual in a frame from a video source. This technology compares selected facial features from an input image with faces in a database.

Retail Image Recognition

This application provides retailers with an understanding of the layout of goods on the shelf. Algorithms process product data in real time to detect whether goods are present or absent on the shelf. If a product is absent, they can identify the cause, alert the merchandiser, and recommend solutions for the corresponding part of the supply chain.

Image Segmentation with Deep Learning in the Real World

In this article we explained the basics of modern image segmentation, which is powered by deep learning architectures like CNN and FCNN. When you start working on computer vision projects and using deep learning frameworks like TensorFlow, Keras and PyTorch to run and fine-tune these models, you'll run into some practical challenges:

Tracking experiment source code, configuration and hyperparameters Convolutional networks have many variations that can impact performance. You'll run many experiments to discover the hyperparameters that provide the best performance for your problem. Organizing, tracking and sharing experiment data can be a challenge.

Scaling experiments on-premise or in the cloud CNNs require a lot of computing power, so to run large numbers of experiments you'll need to scale up across multiple machines. Provisioning machines and setting them up to run deep learning projects is time-consuming; manually running experiments results in idle time and wasted resources.

Manage training data Computer vision projects use training sets with rich media like images or video. A dataset can weigh anywhere from Gigabytes to Petabytes. You need to copy and re-copy this data to each training machine, which takes time and hurts productivity.

Neural Network Concepts

In this article you will learn:

What is a Recurrent Neural Network (RNN)—a Neural Network Model for Language Processing and Sequential Data

A Recurrent Neural Network (RNN) is an algorithm that helps neural networks deal with the complex problem of analyzing input data that is sequential in nature. For example, text written in English, a recording of speech, or a video, has multiple events that occur one after the other, and understanding each of them requires understanding, and remembering, the previous events. While a traditional neural network accepts a set of inputs and analyzes them all together, an RNN network is able to accept a series of inputs, and each time add additional layers of comprehension on top of the previous inputs. For some background on RNNs, see this post by Andrej Karpathy on the 'The Unreasonable Effectiveness of Recurrent Neural Networks'.

What Can RNNs Do?

Language modeling and text generation

A language model captures the structure of a large body of text, and is able to write additional text in the same tone or style. Models work by predicting the most suitable next character, next word, or next phrase in the generated text.

Machine translation

The input is a sequence of words in the source language, and the model attempts to generate matching text in the target language. The input phrase or sentence is fed to the network as a batch, from start to finish, because translation is usually not word-for-word.

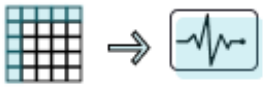
Speech recognition

The input is an audio recording, parsed into acoustic signals, and the model outputs the most likely syllable or phonetic element matching each part of the recording.

Generating image description

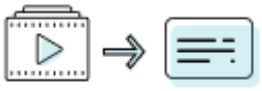
The input is an image, and the model identifies important features in the image and generates a brief text that describes the image.

Time-series anomaly detection



The input is a sequential data series, such as behavior of a user on a network over a full month. The model predicts which of the data in the series represents an anomaly compared to the normal flow of events.

Video tagging

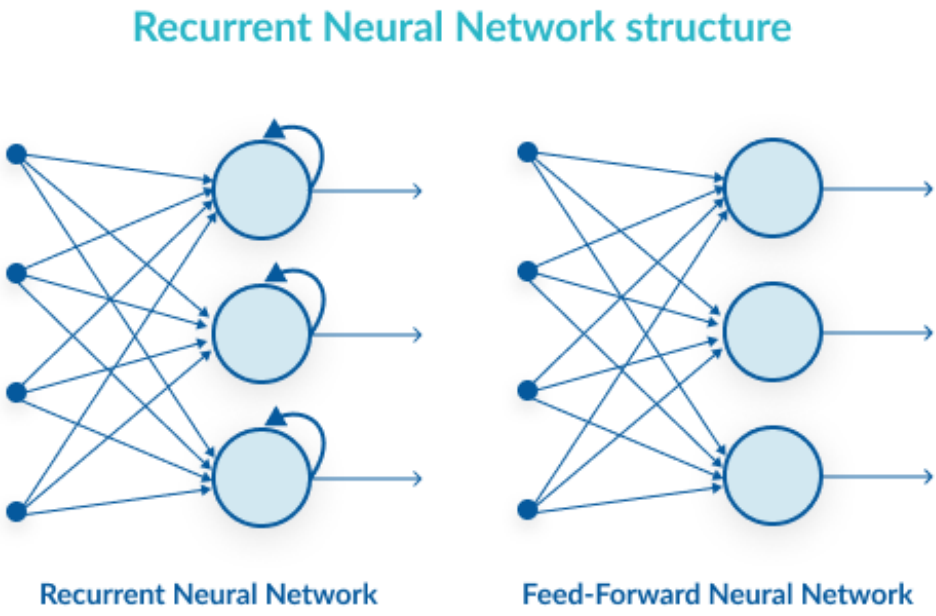


The input is a series of video frames, and the model generates a textual description of what is happening in the video, frame by frame.

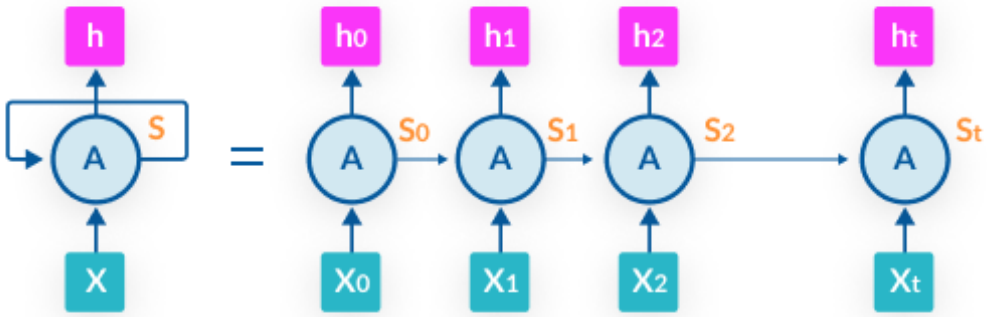
RNN Deep Learning—the Basic Structure

The Neuron Loop

The simplest form of RNN ("vanilla" RNN) is similar to a regular neural network, only it contains a loop that allows the model to carry forward results from previous neuron layers.



The image below "unrolls" how the loop works. The network looks at a series of inputs over time, X0, X1, X2, until Xt. For example, this could be a sequence of words in a sentence. The neural network has one layer of neurons for each input (in our example, one layer for each word).



Unrolled Recurrent Neural Network

At each stage in the sequence, each layer of neurons generates two things:

- **An output—H_0, H_1, H_2**—this is the model's prediction for what should be the next element in the sequence
- **A hidden state—S_0, S_1, S_2**—this is the network's "short-term memory". The hidden state is an activation function, which takes as its input both the hidden state of the previous step and the output of the current step. This allows the model to carry over information from previous steps to the current step.

When the network is trained, it not only assigns weights to each neuron's inputs, like in a traditional neural network but also discovers the parameters of the hidden state function. These parameters define how much of the information from the previous steps should be carried forward to each subsequent step (this may differ between problems).

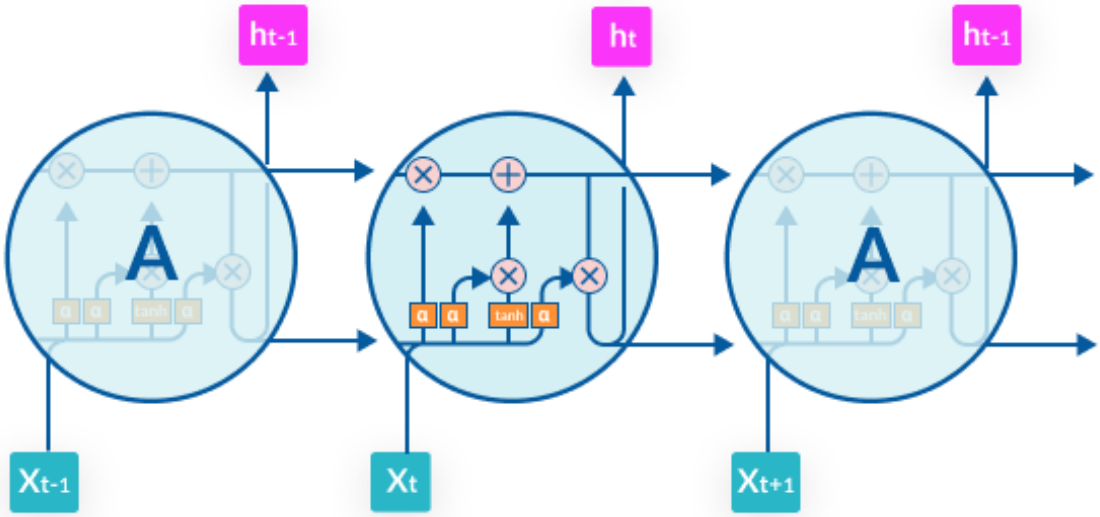
Backpropagation Through Time (BPTT)

Regular backpropagation tweaks the weights of the model in several iterations, using training samples. In an RNN, backpropagation needs to take into account that the model is carrying forward information from each neural layer to the next, and fine tune the weights that govern this "short-term memory". This is called Backpropagation Through Time (BPTT). BPTT "unrolls" the network, with each time step connecting to all the neurons in the next time step (the next hidden neural layer). BPTT uses the chain rule to go back from the latest time step to the previous step, and then to the next-previous, each time using gradient descent to discover the best weights for each neuron and for the hidden state function.

Long Short-Term Memory—the Structure of an LSTM Cell

In the late 1990s, German researchers Hochreiter and Schmidhuber proposed the concept of Long Short-Term Memory (LSTM), which would help an RNN retain information over a longer period of time, not just in between two steps in time. For example, a character's name, used at the beginning of a paragraph in a novel, can be

important for understanding who is speaking or what is being said at the end. An LSTM cell allows the network to pick up such pertinent information and save it, injecting it back into the model when necessary.



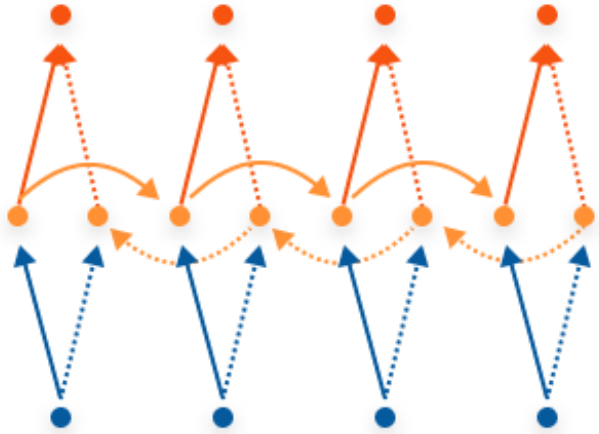
Recurrent Neural Networks: LSTM cells

While a basic RNN has only a simple activation function, an LSTM cell has four. Three sigmoid activation functions output numbers between 0 and 1, and lead to a pointwise multiplication gate. This gate determines whether information should enter the cell or not—0 means no information enters, and 1 means all the information enters. These three cells are used to save pertinent information for later stages of the learning process. In the training process, the network learns what are the optimal values for the gates, or how much of the information should be retained to help the network make the most accurate prediction.

RNN Extensions and Types

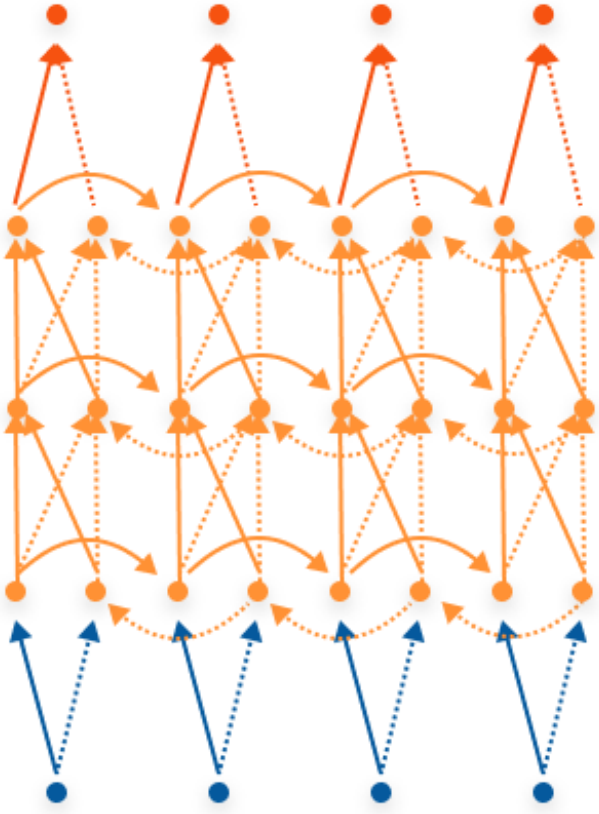
Bidirectional RNNs A bidirectional RNN assumes that the correct output not only depends on the previous inputs in the time series but also on future inputs. For example, in translation models, it is often necessary to use a word from the end of the source sentence in order to predict a word early in the target sentence. To make this possible, two RNNs are stacked on top of each other, one going from beginning to end and the other from end to beginning, and the output is computed based on hidden states of both networks.

Bidirectional Recurrent Neural Network



Deep (Bidirectional) RNNs A deep RNN uses multiple neuron layers per time step. This is, in essence, a three-dimensional neural network which progresses from left to right, from right to left, and from bottom to top. This architecture provides higher learning capacity, but also requires more training data.

Deep (Bidirectional) Recurrent Neural Network



Long Short-Term Memory Units (LSTMs) RNNs An LSTM RNN leveraging an LSTM cell, explained above, has the same architecture as a basic RNN (can be since RNN, bidirectional or deep bi-directional), except that all its activation functions are replaced with an LSTM cell.

Additional RNN extensions and types:

- **Fully recurrent** – a network comprised of layers including neuron-like nodes, where each node is connected to a node in the next layer with a one-way connection.
- **Elman/Jordan networks, or Simple Recurrent Networks (SRN)**:
    - Elman networks are comprised of three layers arranged horizontally as x, y, and z. In addition, a set of "context units" connect to a middle (hidden) layer and are fixed with a weight of one.
    - Jordan networks are similar to Elman networks, but the context units are fed from the output layer instead of the hidden layer.
- **Recursive neural network** – in a recursive network, the same set of weights is applied recursively over a differentiable graph-like structure by traversing the structure in topological order.
- **Bidirectional Associative Memory (BAM)** – a variant of a Hopfield network including two layers, and storing associative data as vectors. Each layer can be used as an input to recall an association and produce an output on the other layer.
- **Hopfield** – an RNN in that includes only symmetric connections. This RNN does not process sequences of patterns and guarantees that it will converge.
- **Hierarchical** – a network in which the neurons are connected in various ways, decomposing hierarchical behavior into useful subprograms.
- **Echo State Network (ESN)** – includes a random hidden layer that is sparsely connected. The only part of the network that can be trained is the weights of the output neurons.
- **Neural Turing machines (NTMs)** – refers to the extension of recurrent neural networks by coupling them to external memory resources.
- **Differentiable Neural Computer (DNC)** – extend Neural Turing machines, using chronology records and fuzzy amounts of each memory address.
- **Recurrent Multi-Layer Perceptron (RMLP)** – comprised of cascaded subnetworks, where each one contains multiple layers of nodes and acts as feed-forward (except for the last layer, which may include feedback connections).
- **Independent RNN (IndRNN)** – addressing the problem of gradient vanishing and exploding in fully-connected RNNs, with INdRNNs, neurons are independent of each other's history.
- **Neural history compressor** – an unsupervised stack of RNNs, which learns to predict the next input using previous inputs.
- **Second order RNNs** – this network uses higher order weights to allow a direct mapping to a finite state machine.
- **Gated recurrent unit (GRU)** – a gating mechanism in recurrent neural networks introduced in 2014, which includes fewer parameters than LSTM.

- **Continuous-Time Recurrent Neural Network (CTRNN)** – this network models the effects of an incoming spike train using a system of differential equations.
- **Multiple Timescales Recurrent Neural Network (MTRNN)** – this neural-based model uses self-organization to simulate the functional hierarchy of the brain.
- **Neural Network Pushdown Automata (NNPDA)** – similar to NTMs, but instead of tapes, differentiable and trained analog stacks are used.

RNN in the Real World

Relational Neural Networks (RNN) are the state of the art algorithm used to perform deep learning on language and sequential data. To create an RNN in a real-world project, you'll fire up a deep learning framework such as Tensorflow or Keras, select hyperparameters such as network architecture, number of layers and an activation function, and the model will immediately be ready to train. For example, in TensorFlow, you can choose an optimizer for the weights (simple Stochastic Gradient Descent, AdaOptimizer, MomentumOptimizer, etc), an activation cell (Basic, Gru, LSTM, Multi RNN), and an RNN architecture (static RNN with a uniform length for all input sequences, dynamic RNN with the ability to have inputs of different lengths, and static bidirectional RNN). Today's deep learning frameworks let you create Recurrent Neural Networks quickly with just a few lines of code, despite their theoretical complexity. However, in real-world projects you might run into a few challenges:

Tracking experiment progress, source code, and hyperparameters across multiple RNNs.

Running experiments across multiple machines—for bidirectional RNNs operating on long data sequences, or RNNs processing video and audio, real projects will likely require running experiments on multiple machines. Provisioning these machines, configuring them and distributing the work among them can become a burden.

Manage training data—some RNN projects have a simple training set involving one or more text files. But for RNNs processing rich media, training sets can weigh Gigabytes or more, and moving data between training machines is difficult and time-consuming.

CapsNet: Origin, Characteristics, and Advantages

missinglink.ai/guides/neural-network-concepts/capsnet-origin-characteristics-advantages

A capsule network is a neural network architecture developed by Stanford scientist Geoffrey Hinton. In the capsule network, a distinct approach is applied to image processing, which includes equivariant mapping and the mapping of the hierarchy of parts. Equivariant mapping supports the preservation of position and pose information. The mapping of the hierarchy of parts assigns each part to a whole. This information provides an understanding of objects, from a three-dimensional context. In this page, you will learn more about the capsule network, understand what is a capsule, and how a CapsNet differs from a traditional convolutional neural network (CNN). **In this page:**

- **A brief history**
- **What are capsule neural networks**
- **The need for CapsNets in image classification**
- **Drawbacks of CNNs**
- **How capsule networks solve these problems**
- **How MissingLink can help scale up CapsNet**

A Brief History

Geoffry Hinton and his team, first introduced the concept of CapsNets in 2011, in a paper titled “Transforming Autoencoders”. However, it was only in 2017, that Hinton and his team developed a dynamic routing mechanism for capsule networks. This process was believed to decrease error rates on MNIST and to decrease training set sizes. Results were deemed to be of higher quality than a Convolutional Neural Network (CNN) on highly overlapped digits.

What Are Capsule Neural Networks? (CapsNet)

A Capsule Neural Network (CapsNet) is a type of Artificial Neural Network (ANN). The approach, which aims to mimic biological neural organizations, can be used to model hierarchical relationships. The concept involves adding structures known as capsules to a convolutional neural network. **Traditional CNNs** In traditional CNNs, every neuron in the first layer corresponds to a pixel. It then feeds information about the pixel to the next layers. The next convolutional layers gather a group of neurons so that an individual neuron can represent a whole group of neurons. A CNN can thus learn to represent a group of pixels that look like, for example, the eye of a cat, particularly if we have several examples of cat eyes in our data set. The neural network will learn to increase the weight (importance) of that eye neuron feature when determining if that image is of a cat. **The role of capsules** This approach only considers the existence of the object in the image around a specific location. It does not care about the direction of the object or spatial relations. Capsules, however, contain more information about individual objects. A

capsule is a vector. Capsules denote the features of the object and its probability. These features can include parameters such as “pose” (size, position, orientation), velocity, deformation, albedo (light reflection), texture, and hue.

The Need for CapsNets in Image Classification

Although convolutional networks successfully implement computer vision tasks, including localization, classification, object detection, instance segmentation or semantic segmentation, the need for CapsNets in image classification arises because:

- **CNNs are trained on large numbers of images** (or reuse parts of neural networks that have been trained). CapsNets generalize effectively and require less training data.
- **CNNs can’t deal with ambiguity well.** CapsNets do, so they can make sense of crowded scenes. However, they currently struggle with backgrounds.
- **CNNs need additional components** to automatically recognize which object a part belongs to, for example, this arm belongs to this person. CapsNets provide a hierarchy of parts.

Drawbacks of CNNs

CNNs are not good with changes in object orientation If you turn an image of a face upside down, the network won’t be able to identify the nose, eyes, mouth, and the spatial connection between these features. Furthermore, if you switch the location of the eyes and nose, the CNN network will identify the face, although it is not a “real” face. CNNs can learn statistical patterns in images, but they can’t learn what makes something look like a face. **CNNs lose information in the pooling layers** When performing pooling, CNNs tend to lose information that is useful for performing tasks such as object detection and image segmentation. When the pooling layer loses necessary spatial information about location, rotation, scale, and different positional characteristics of the object, the task of object segmentation and detection becomes challenging. Today, CNN architects can reconstruct positional information, via advanced techniques. However, these techniques are not entirely accurate, and reconstruction is an involved process. An additional issue with the pooling layer is that if the position of the object is slightly altered, the activation doesn’t change with its proportion. The result is accurate in relation to image classification, however, it can make it hard to precisely locate an object in an image.

How Capsule Networks Solve These Problems

Capsule networks work by implementing groups of neurons that encode spatial information as well as the likelihood of an object existing in an image. The length of a capsule vector is the likelihood that a feature is present in the image, and the direction of the vector corresponds with its pose information. A capsule is a group of neurons whose activity vector corresponds to the instantiation parameters of an object or object

part. We use the length of the activity vector to show the likelihood that the entity is present. The orientation of the vector indicates the instantiation parameters. In computer graphics functions like rendering and design, the CNN network typically generates objects by giving them a parameter which will render the form. In capsule networks, however, the opposite occurs. The network learns how to inversely render an image by examining the image and attempting to predict what the instantiation parameters are for the image. The network learns to predict this by attempting to reproduce the object it believes it detected and contrasting it against the labeled example from the training data. It thus gets better at predicting the instantiation parameters. **Addressing the "Picasso problem"** CapsNets can address the "Picasso problem" in image recognition: images that show the right component but they are not in the right spatial relationships, for example, if in a "face" the location of the eye and ear are swapped. While viewpoint changes have nonlinear effects at the pixel level, they have linear effects at the object or part level, and CapsNets use this. This is like inverting the rendering of an object with several parts. **Viewpoint invariance** A traditional CNN can only identify a cat face based on similar cat face detectors stored within the training dataset, with similar size and orientation, as the features of the cat face are kept in locations inside the pixel frame. For example, it may have a picture of a cat face where the nose is around pixels [60, 60], the mouth is around [60, 30], and the eyes around [30, 80] and [80, 80]. Thus, it can only identify images that have similar features in similar locations. Therefore, it needs to have a distinct representation for a "cat face rotated by 30 degrees" or a "small cat face". Those representations would ultimately map to the same class, but it means the CNN must see many examples of each transformation type to be able to recognize a cat face in the future. A capsule network can develop a general representation of a "cat face" and see the transformation (size, rotation, etc.) of each of its features (mouth, snout, etc.). It can tell if all the features are transformed or rotated in the same direction and to the same degree. It can, therefore, predict with more confidence that it is a cat face. CapsNets generalize the class, rather than memorizing each single viewpoint variant, so it is not restricted to a specific viewpoint.

How MissingLink Can Help Scale Up CapsNet

Shortcomings of capsule networks:

- Insufficient testing for large image datasets (e.g. ImageNet) doesn't permit making informed conclusions about the further likelihood
- Higher complexity of computing and implementation and computing demanded

Use the MissingLink deep learning framework to:

- **Scale out CapsNet** automatically on many machines, both on-premise or in the cloud
- **Specify a cluster of machines** and automatically run deep learning jobs, while ensuring maximum utilization of resources
- **Prevent wasted time** by immediately running experiments consecutively and

shutting down cloud machines efficiently once jobs are finished.

With MissingLink you can also control a large number of experiments, share and track results, oversee large data sets and sync them seamlessly to training machines. **Learn more about MissingLink.ai and discover how simple it is!**