Chapter Title: Fundamentals of Artificial Intelligence and Machine Learning: How Neural Nets Work
Author: Yalong Pi, Associate Research Scientist, Texas A&M Institute of Data Science, Texas A&M University

Abstract
Artificial Intelligence (AI) and Machine Learning (ML) are increasingly transforming society along with the advancing computing technology and vast data generated in the information age. Yet, due to its mathematics and coding requirements, existing technical literature often poses barriers to non-computer science professionals. This chapter offers a foundational overview of AI and ML tailored to an audience population that is not STEM. It introduces the core mathematical concepts with intuitive explanations of neural networks, which is the common foundation of all current AI and ML systems, ranging from image and video understanding, fast protein structure prediction, natural language processing, and recommendation systems. By using simple examples and tools (such as an Excel spreadsheet), the chapter aims to deliver a meaningful understanding of many key AI and ML concepts, including dataset preparation, model/algorithm training, validation, testing, over-fitting, performance metrics, activation function, confusion matrix, and loss function. With these concepts in mind, several state-of-the-art AI and ML systems will be reviewed at a high level.

1. Introduction

There is a collection of established statistical tools developed over time using mathematical properties to analyze big datasets. For instance, linear or polynomial regression for data modeling and prediction. Principal component analysis (PCA) is designed to reduce the data dimensions using eigenvalues and eigenvectors. With enough data, decision tree models such as random forest and XGboost can be used for regression and classification problems. Clustering tools include k-nearest neighbors, k-means clustering, DBSCAN, and others, which do not require data labels to group data. These methods, supervised or unsupervised, are well covered in existing data science materials and courses.

The advancing AI and ML are based on a technique called neural networks to model the high-dimensional mathematical relationships between digital inputs and outputs. Jeffery Hinton introduced the foundational tool to optimize/tweak such networks (Rumelhart et al., 1986) which describes multiple layers, activation functions, and backpropagation. Later, they enabled a drastic advancement to deep neural networks (DNNs) with many layers (Krizhevsky et al., 2012). However, many techniques and concepts used in establishing such DNNs are often challenging topics for audiences that are not in the mathematics or computer science domains.

First, a toy example with actual numbers will be used to build a simple neural network to show the basic setups to approximate a linear function to predict the value y given value x. This is to show an intuitive understanding of the basic components and get a quick start with machine learning. Next, key concepts including training, validation, testing, activation function, loss function, model deployment, and evaluation will be discussed. Third, equipped with the fundamentals, more complex problems, including computer vision classification, detection, segmentation, and object tracking, will be illustrated at a high level. The details and limitations of the current generative AI and natural language processing will be discussed.

2. A Toy Example with Gradient Descent

Programming languages such as Python are often required to learn and develop DNNs in classrooms and workshops because popular libraries such as PyTorch and TensorFlow can support rapid experiments in a handful of lines of code. While these are useful, working mathematics and calculations are often briefly mentioned as training and learning processes without showing the mechanism. Because these libraries are developed for computer science applications, many functions are packaged and non-breakable, making it difficult to show various parts of the DNN. This section shows how to build a neural network using Microsoft Excel or Google Sheets without third-party plugins. The purpose is to illustrate the internal calculation step by step to enhance the intuitive understanding of ML and AI. This method is for educational purposes and cannot replace established packages such as PyTorch or TensorFlow.

The toy dataset in Table 1 represents three inputs x, paired with output y data points in each row. This example is consistent in this chapter, explaining many details. It is a well-studied problem using linear regression with the least squares method to approximate such an input-output relationship. The goal of such regression is to find the parameters a (slope) and b (interception), which produce $y_{pred} = ax + b$. For each input x, the predicted value y_{pred} will be compared with each y_{true} in Table 1. The best-fitting line is described by a and b.

Table 1. Toy example with input x and output y. Here, y has a sub-note true, indicating this is the data for the target y values.

x	y_{true}
1	10
2	20
5	30

Instead of linear regression with two parameters, a neural network is constructed as shown in Figure 1 to approximate the relationship between x and y, such that given another new x, the network can predict the y value with some accuracy. It will become clear that neural networks can approximate non-linear relationships with activation functions, which makes them very versatile in many domain applications. However, current mathematical knowledge cannot prove why neural networks are so effective or why they work at all. The mathematical proof for neural networks and activation functions is an actively researched area (Daubechies et al., 2022).

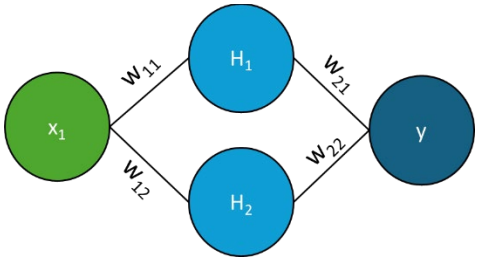


Figure 1. Simple neural network.

The neural network example in Figure 1 is straightforward. Each circle is one number holder, a.k.a. neurons in literature (LeCun et al., 2015). Numbers x_1, H_1, H_2, y are calculated using the following equations 1 to 3. This process is also easy to set up in Microsoft Excel or Google Sheets for a quick test.

It is obvious that the successful y_{pred} depends on the parameters $w_{11}, w_{12}, w_{21}, w_{22}$, represented by solid lines. They are named w because the output is the weighted sum of the input. With randomly initialized weights, we can get the forward pass results and save the intermediate output H_1 and H_2.

$$H_1 = x_1 * w_{11} \quad (1)$$
$$H_2 = x_1 * w_{12} \quad (2)$$
$$y_{pred} = x_1 * w_{11} * w_{21} + x_1 * w_{12} * w_{22} \quad (3)$$

We cannot manually try out all possible weights to find the best fit. Here, we need the method called gradient descent, which Augustin-Louis Cauchy discovered. After randomly initializing the weights, we need to measure how off the current guess y_{pred} is off from y_{true} and make changes accordingly. For example, we can use a loss function of square error, as in Equation 4, to measure the error for each row and sum the losses together. This way, it does not matter why y goes first. It is worth noting that loss functions work with individual data pairs (e.g., Table 1) and batches (e.g., many tables). For a batch with n samples, the batch loss is the value of averaging all sample losses.

$$loss = \left(y_{true} - y_{pred}\right)^2 \quad (4)$$

Now, we will look at the core of all machine learning systems: How to train a model. Here, we take w_{22} in the first row in Table 1 as an example, and setting $w_{11} = w_{12} = w_{21} = 1$. This is different from regular calculus textbooks because the goal is to provide an intuitive understanding. Later, we will formalize this with the proper notations. Plug the numbers $y_{true} = 10$ and $x_1 = 1$ we have $loss = 91 - 18w_{22} + w_{22}^2$. The detailed steps are in Equation 5 to help understand.

$$\begin{aligned} loss &= (y_{true} - x_1 * w_{11} * w_{21} + x_1 * w_{12} * w_{22})^2 \\ &= \left(10 - (1 * 1 * 1 + 1 * 1 * w_{22})\right)^2 \\ &= \left(10 - (1 + w_{22})\right)^2 \\ &= (9 - w_{22})^2 \\ &= 91 - 18w_{22} + w_{22}^2 \end{aligned} \quad (5)$$

Plotting Equation 5 with loss on the vertical and w_{22} on the horizontal axis, shows the following graph in Figure 2. This is the parabola graph, which should be covered in basic math classes in high school. The slope can be calculated using the derivative of this loss function: $loss' = \frac{loss(w_{22}+\Delta) - loss(w_{22})}{\Delta} = 2 * w_{22} - 18$ when Δ is approaching zero. Importantly, the loss graph is symmetrical to the line of $w_{22} = 9$. In other words, when w_{22} is on the right side of this line, the slope of the function is positive and negative on the left side. This key feature allows us to make the following update: the new weight $w_{22}(new) = w_{22}(old) - loss' * LR$, where LR is a very small learning rate, for example, 0.0001, w_{22} (old) is the old weight.

When the old w_{22} is on the right side, the slope is positive, hence change it with $-loss' * LR$ move the weight toward left, resulting in smaller loss. On the other hand, when the old weight w_{22} is on the left, the change $-loss' * LR$ is positive, moving the weight to the right with smaller loss. Altogether,

$w_{22}(new) = w_{22}(old) - loss' * LR$ method constantly changes the weights so the loss is smaller. This update happens in many iterations until the minimum loss is reached. The red arrows in Figure 2 represent each step above.

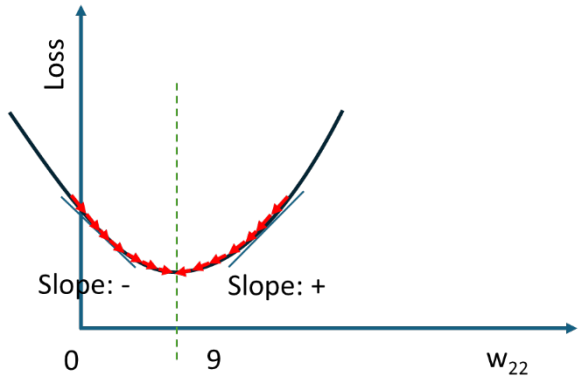


Figure 2. One-dimensional gradient descent for the function $loss = 91 - 18w_{22} + w_{22}^2$. The weight is updated by: The new weight $w_{22}(new) = w_{22}(old) - loss' * LR$, where LR is a very small learning rate, for example, 0.0001, w_{22} (old) is the old weight.

Now, consider another weight w_{11} that adds one more dimension to the function. The curve in Figure 2 will become a surface in Figure 3. Each iteration, the update for w_{22} and w_{11} happen simultaneously, marked with green and red colors along their respective direction. The underlying mechanism for the update steps is the same as the 2-dimensional. Follow the same principle, the weight can be added to this high-dimensional space and updated together. However, it is impossible to visualize all the dimensions.

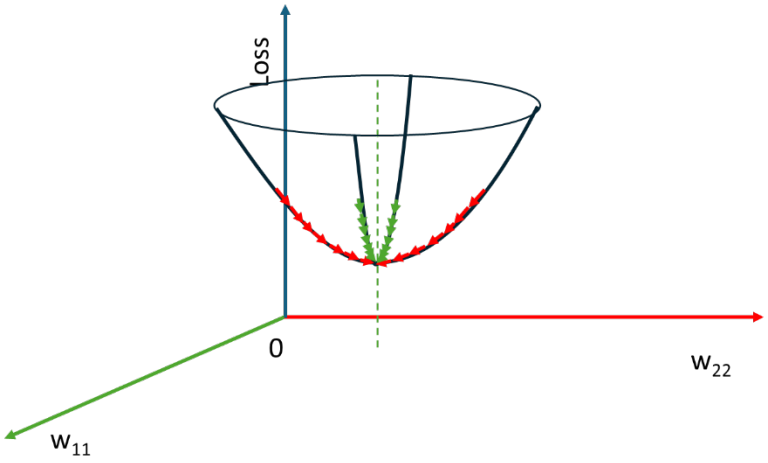


Figure 3. High-dimensional gradient descent. The new weight update follows: $\overrightarrow{w\ (new)} = \overrightarrow{w\ (old)} - \nabla * LR$.

Formally, the rate of change of loss with respect to each weight is called the partial derivative, written as $\frac{\partial loss}{\partial w_{ij}}$. The update scheme for all four weights is expressed in Equations 6 to 9, which is the same as above. It is troublesome to write out all the partial derivatives, especially when there are many weights, so we group all w into $\vec{w}$. Now, the update mechanism can be summarized in Equation 10. This equation will appear many times in this chapter, and it is important to know that the dimension of ∇ has to match that of

$\vec{w}$ to allow the update. This process is often called backpropagation, training, model fitting, learning, or optimization in different literature.

$$w_{22}(new) = w_{22}(old) - \frac{\partial loss}{\partial w_{22}} * LR \quad (6)$$

$$w_{11}(new) = w_{11}(old) - \frac{\partial loss}{\partial w_{11}} * LR \quad (7)$$

$$w_{12}(new) = w_{12}(old) - \frac{\partial loss}{\partial w_{12}} * LR \quad (8)$$

$$w_{21}(new) = w_{21}(old) - \frac{\partial loss}{\partial w_{21}} * LR \quad (9)$$

$$\overrightarrow{w\,(new)} = \overrightarrow{w\,(old)} - \nabla * LR \quad (10)$$

3. Concepts with implementation details

3.1 Dataset

Table 1 is the full dataset with all three input and output pairs. The ground truth outputs are sometimes called labels or targets. When the dataset size is big enough, a small portion will be preserved for unseen testing, hence named the testing dataset. The remaining pairs will be further divided into training and validation datasets. In many AI and ML applications, dataset establishment is critical and requires the most effort to set the stage for the algorithm.

3.2 Neural Network Algorithm

The neuron connections and input-output setup in Figure 1, in combination with the step-by-step mathematical calculations in Equations 1 to 10, is the neural network algorithm. To solve complex problems, such as image recognition, the algorithm will consist of millions of neurons and be structured for specific tasks. In such cases, the neural network is named a deep neural network, and the learning process is called deep learning. There are two distinct dataset processes of running the algorithm: forward pass and backpropagation.

3.3 Forward pass

With randomized initial values for $w_{11}, w_{12}, w_{21},$ and w_{22}, the intermediate values H_1 and H_2 are computed and passed forward to calculate each y_{pred} based on the input x_1. It is straightforward in Equation 3. This forward pass process is called neural network model inference or deployment: Predicting the output values based on the input values. It is clear that all input and output data must be numerical values to allow the forward pass. There is no information about the ground truth used in the forward pass.

3.4 Backpropagation

The weight update details are listed below in Equations 11 to 14, where the partial derivative for each weight is spelled out using Equations 3 and 4. By applying the chain rule, the intermediate values H_1 and H_2 are used to compute the gradient without additional computational cost. The chain rule is an advanced concept in calculus that might not be familiar to non-STEM students. The key here is understanding that partial derivatives can be computed using the forward pass result. Packages such as Pytorch carry out this process, but it is possible to set this up in Excel when the network is small. With large neural nets, it is not

possible to handle millions of gradients without computers. It is also clear that defining the loss function to be easily differentiable is an advantage.

$$\frac{\partial loss}{\partial w_{22}} = \frac{\partial loss}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_{22}} = 2 * \left(y_{pred} - y_{true}\right) * H_2 \quad (11)$$

$$\frac{\partial loss}{\partial w_{11}} = \frac{\partial loss}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_{11}} = 2 * \left(y_{pred} - y_{true}\right) * w_{21} * x_1 \quad (12)$$

$$\frac{\partial loss}{\partial w_{12}} = \frac{\partial loss}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_{12}} = 2 * \left(y_{pred} - y_{true}\right) * w_{22} * x_1 \quad (13)$$

$$\frac{\partial loss}{\partial w_{21}} = \frac{\partial loss}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_{21}} = 2 * \left(y_{pred} - y_{true}\right) * H_1 \quad (14)$$

3.5 Batch

Table 2 shows the setup for the Excel neural network. For simplicity, we treat each epoch as a batch, i.e., a group of data points to be processed before updating the weights. The batch gradient is the average of the gradients over all examples, which will be used to update the weights in the next epoch. It is also common for some literature to use the gradient summation.

3.6 Excel implementation

For each data input-output pair, the y_{true} and x_1 values are ready. Next, the forward pass produces y_{pred}. The byproduct of this process is $w_{11}, w_{12}, w_{21},$ and w_{22}, and the intermediate values H_1 and H_2. Using Equations 11 to 14, the gradient and batch gradient (average gradient of three data points) are calculated. In the next epoch, instead of randomly assigning $w_{11}, w_{12}, w_{21},$ and w_{22}, they are updated using Equations 6 to 9. In other words: $\overrightarrow{w\ (new)} = \overrightarrow{w\ (old)} - \nabla * LR$. After setting up the rules in epochs 1 and 2, the key is to duplicate these functions to repeat many times. Along the way, loss and batch loss (average of three losses) are recorded and plotted in Figure 4.

Table 2. Excel example with the first two epochs. The key is the second epoch, which sets the update mechanism with the gradients from the previous step. This will be repeated in the following epochs.

epoch	1			2			Repeat
x	1	2	5	1	2	5	…
y_{true}	10	20	30	10	20	30	…
w_{11}	0.3			Eq. 7			…
w_{12}	0.1			Eq. 8			…
H_1	0.3	0.6	1.5	0.41388	0.82776	2.0694	…
H_2	0.1	0.2	0.5	0.188573	0.377147	0.942867	…
w_{21}	0.9			Eq. 9			…
w_{22}	0.7			Eq. 6			…
y_{pred}	0.34	0.68	1.7	0.52259	1.045181	2.612951	…
$loss$	93.6	373.2624	800.89	89.82129	359.2852	750.0504	…
$\frac{\partial loss}{\partial w_{11}}$	Eq. 12	Eq. 12	Eq. 12	-17.7789	-71.1154	-256.88	…

$\frac{\partial loss}{\partial w_{12}}$	Eq. 13	Eq. 13	Eq. 13	-13.5082	-54.0329	-195.175	…
$\frac{\partial loss}{\partial w_{21}}$	Eq. 14	Eq. 14	Eq. 14	-7.84502	-31.3801	-113.35	…
$\frac{\partial loss}{\partial w_{22}}$	Eq. 11	Eq. 11	Eq. 11	-3.57437	-14.2975	-51.6447	…
batch gradient for w_{11}	-113.88			-115.2579574			…
batch gradient for w_{12}	-88.57333333			-87.57193007			…
batch gradient for w_{21}	-37.96			-50.85820654			…
batch gradient for w_{22}	-12.65333333			-23.17217922			…
batch loss	422.4893333			399.7189663			…

Altogether, we can see the training progress with batch losses in Figure 4 with 20 epochs. The batch loss decreases over epochs, indicating successful gradient descent. We can deploy the model using the latest weights at epoch 20 and Equation 3 to predict any input value we are interested in. It is important to note that activation functions are not used, i.e., this is still a linear approximation.

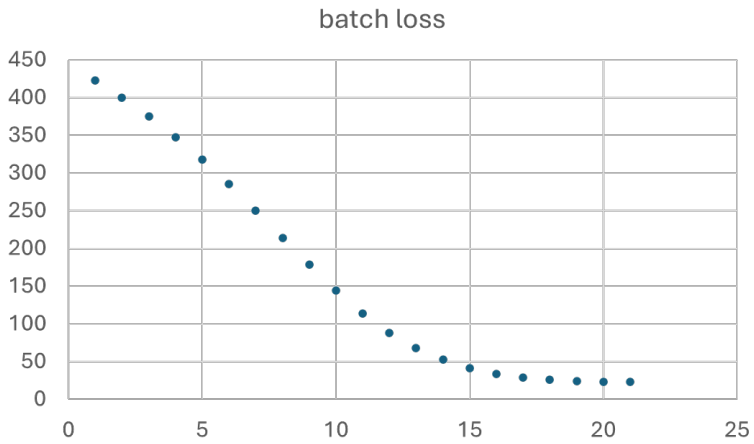


Figure 4. Training batch loss over epoch plot.

3.7 Loss function

In the example above, the loss function in Equation 4 was used to measure the prediction error and update the weights. Because the example neural network predicts the output value directly, the regression discrepancy is easily captured by the mean square error method. It is important to note that this is not the only loss function format. As long as the loss function is differentiable, the neural network can learn from the input and output pair. This means that it is possible to design a wide variety of output formats and measure the error with the matching loss functions. This feature makes neural network systems highly adaptable to almost all numerical tasks.

Beyond regression, there is a critical class of loss functions that are particularly useful for classification tasks: the negative likelihood loss function: $loss = -\log(y_{pred})$. Because of the nature of classification tasks, the output y_{pred} is often normalized (e.g., SoftMax) between 0 and 1, representing probabilities for

each class. For example, Figure 5 shows a bigger network than the example with output numbers: y_1 and y_2 indicating the probabilities for each class 1 and 2. This is a fully connected neural network with 15 weights (solid lines) in the first layer, 25 in the second layer, and 10 in the last layer. It is a helpful exercise to count the number of weights to check the network dimension.

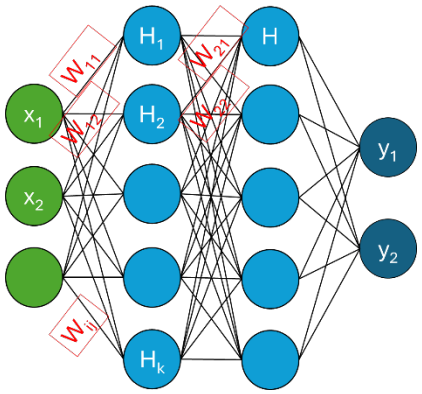


Figure 5. A bigger neural network structure with three inputs and two outputs. The weight update follows:

$$\overrightarrow{w\ (new)} = \overrightarrow{w\ (old)} - \nabla * LR$$

The ground truth will be arranged so that a positive case is equal to 1 and a negative case is equal to 0. A straightforward example could be $y_{1\_true} = 1$, $y_{2\_true} = 0$ with the prediction being $y_{1\_pred} = 0.95$, $y_{2\_pred} = 0.05$. One common way to measure the error between the predicted and ground truth probabilities is to consider only the positive ground truth cases and sum up their negative likelihoods. This loss has a special name: cross-entropy, plotted in Figure 6. When y_{pred} is close to 100%, the loss value is small (close to zero), indicating a very small weight change while training. This is to say, do not make significant changes in weight because the model produces good quality predictions. On the other hand, when y_{pred} is a small value, i.e., low probability for a positive case, a larger loss will be computed and therefore a higher gradient to make more significant changes in the weights.

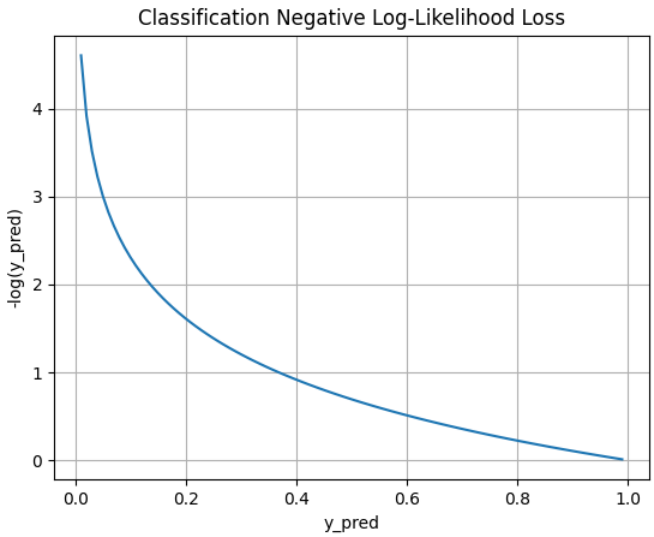


Figure 6. Negative likelihood loss (vertical) vs prediction probability (horizontal) for one class.

There are specialized cross-entropy losses, such as binary cross-entropy, which also consider the negative cases and add them to the gradients. In many advanced ML and AL problems, a combined loss function of regression and classification is often designed with dynamic calculations. Nevertheless, the training process remains the same as described above.

3.8 Activation function

The neural networks we discussed so far are linear combinations of weighted sums. To model complex non-linear tasks, we need to add non-linearity to the models. One efficient way to do this is to use activation functions such as Rectified Linear Unit (RELU), which keep the intermediate value H if it is positive, otherwise $H = 0$. This is one efficient way to make the neural network work with a non-linear problem, adding no additional computational cost. Looking back at the Excel machine learning example, it is a linear function, which can be shown by plotting a series of predictions with a range of inputs. Adding activation functions will make it non-linear.

3.9 Overfitting and underfitting

After each epoch, it is helpful to plot the training loss values. A decreasing loss indicates the neural network is training properly. Meanwhile, testing the network with the updated weight on the validation dataset (unseen) will give a reasonable estimate of the model's performance on the testing dataset. Because the distribution of the validation dataset differs from the training dataset, the neural network might specialize too much on the training dataset, where the validation performance worsens at some epoch. This tipping point separates the underfitting and overfitting of the model. Standard methods to combat overfitting include early stopping, regularization, random weight initialization, and dropout.

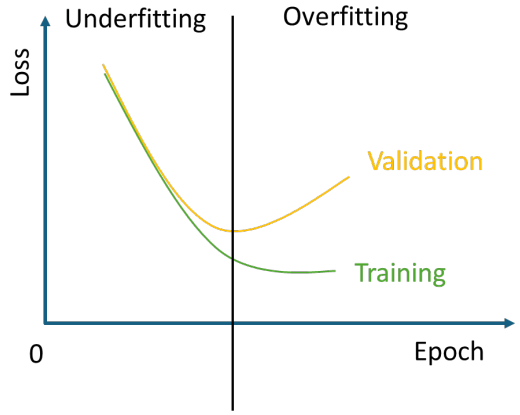


Figure 7. Underfitting and overfitting.

3.10 Performance Metric

The metrics for regression tasks are straightforward: Comparing the prediction and ground truth using various statistical tools, including mean average precision, absolute average precision, square mean error, etc. On the other hand, the performance indicator for classification tasks relies on the binary confusion matrix, as shown in Table 3. Each prediction can be categorized as a true positive (TP) prediction, false positive (FP) prediction, true negative (TN) prediction, and false negative (FN) prediction. By counting how many cases for each category, Equations 15 to 18 compute accuracy, precision, recall, and F1 scores, which are commonly reported in classification problems. All four metrics range from 0 to 100%.

Table 3. Confusion matrix for binary classification.

Prediction\Ground Truth	True	False
Positive	TP	FP
Negative	TN	FN

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (15)$$

$$precision = \frac{TP}{TP + FP} \quad (16)$$

$$recall = \frac{TP}{TP + FN} \quad (17)$$

$$F1 \ = 2 * \frac{precision * recall}{precision + recall} \quad (18)$$

4. Image Processing

Using optical cameras, digital images or videos are captured using pinhole principles and light-sensitive sensors based on the photoelectric effect. Because human eyes are most sensitive to red, green, and blue (RGB) light, common cameras record such light bands, followed by screens display the pixels for each color. Many military, biology, or agriculture applications utilize light bands beyond RGB, such as infrared light. Due to the convention of 8-bit computers, the pixel brightness normally ranges from 0 to 255 ($2^8 - 1$) where 0 is the lowest value corresponding to black color. Figure 6 shows the RGB image input with a small toy dimension of $5 * 5 * 3$ where each x_i indicating the individual pixel brightness intensity.

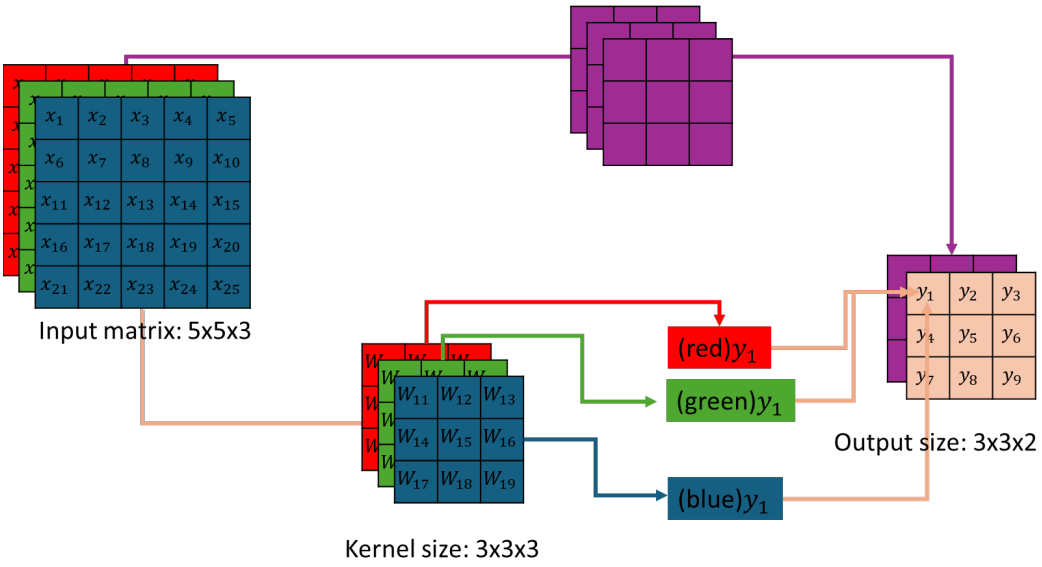


Figure 8. Convolutional computation example.

ML and AI computer vision enable machines to interpret and analyze visual information for tasks such as image classification, object detection, segmentation, and tracking, which were challenging for traditional computer vision hand-engineered techniques. The key convolutional neural networks (CNNs) are designed for matrix-like input, such as images or arrays. The convolutional computation concept illustrated in Figure 8 is a toy example applying two 3 by 3 by 3 convolutional kernels, each of which contains 27 weights, on the $5 * 5 * 3$ input pixel matrix to produce a $3 * 3 * 2$ matrix output.

Breaking down this convolutional computation shows the weighted sum details, which are often confusing to a non-computer audience. Let's focus on the blue channel and apply the kernel at the top-left position of the input to calculate the output $(blue)y_1$. The element-wise computation is detailed in Equation 19 below, which is the exact mechanism as Equation 3 with more weight $\vec{w}$ and inputs $\vec{x}$. At the same position, but for channel green and red, the $(green)y_1$ and $(red)y_1$ are calculated the same way. Altogether, the output $y_1 = (red)y_1 + (green)y_1 + (blue)y_1$. Next, move the kernel position one pixel to the right, and carry out the same computation, which will produce the value for y_2, and so on. This naturally shrinks the input by one pixel. The purple kernel represents a separate set of weights that will produce the second channel in the output. A typical CNN has a few hundred kernels applied in different layers depending on the design.

$$w_{11} * x_1 + w_{12} * x_2 + w_{13} * x_3 + w_{14} * x_6 + \cdots + x_{13} * w_{19} = (blue)y_1 \quad (19)$$

Such convolutional network preserves spatial information, assisting almost all the computer vision models, including ResNet (He et al., 2016) for image classification, YOLO (Redmon et al., 2016) for object detection, U-Net (Ronneberger et al., 2015) for pixel segmentation, and Mask-RCNN for instance detection (He et al., 2017). With prepared labels for each computer vision task, the training mechanism: $\vec{w}\ (new) = \vec{w}\ (old) - \nabla * LR$, is still the same.

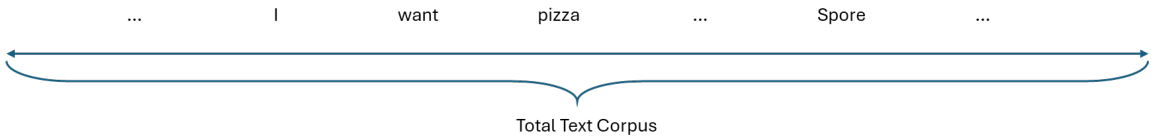
Videos are consecutive frames of digital images recorded and displayed at a frequency per second (FPS) rate. A typical real-time FPS is 30, while some monitoring systems use as low as 10 to save space. Object tracking and understanding based on frame-level masks, bounding boxes, or pixels are vital to many applications, including autonomous vehicles, public monitoring, manufacturing, and more.

5. Natural Language Processing (NLP)

5.1 Tokenization and embedding

Neural networks support most NLP tasks and allow machines to understand, generate, and translate human languages. First, the input text must be converted to numbers for the neural network input. Because of the variety of languages, grammar, and typos, it is helpful to break text input into smaller standard units called tokens (e.g., words with spaces in front). This token definition is not the same as in dictionaries and includes punctuation, unofficial language, and typos. Second, word embedding turns each token into a vector with n numbers. There are many ways to tokenize and embed languages, and it is an active research area. It is worth noting that the tokenization and embedding must be consistent for the same large language model (LLM).

One popular embedding strategy assumes that words that appear together more frequently have similar meanings. Figure 9 shows a toy example where the embedding table on the bottom left contains the n dimensional vectors for each token in the vocabulary (e.g., 40k). Consider all text corpus lined up (e.g., all internet text: 100 billion tokens), the word "I" "want" "pizza" must appear together more frequently than "spore". These words can repeat. In other words, if we plot the first three embedding values in a 3-dimensional space, the data points "I" "want" "pizza" must show up closer than "spore". Although we cannot visualize beyond three dimensions, we can still measure the distance among word embeddings using metrics such as cosine similarity. Currently, popular embedding methods include Word2Vec (Mikolov et al., 2013) and GloVe (Pennington et al., 2014), which all learn embeddings from massive text (e.g., Wikipedia) in an unsupervised manner.



Vocabulary	Embedding 1	Embedding 2	Embedding 3	...	Embedding n
...	...	...	...	...	...
I	e_{i1}	e_{i2}	e_{i3}	...	e_{in}
want	e_{w1}	e_{w2}	e_{w3}	...	e_{wn}
pizza	e_{p1}	e_{p2}	e_{p3}	...	e_{pn}
...	...	...	...	...	...
spore	e_{s1}	e_{s2}	e_{s3}	...	e_{sn}
...	...	...	...	...	...

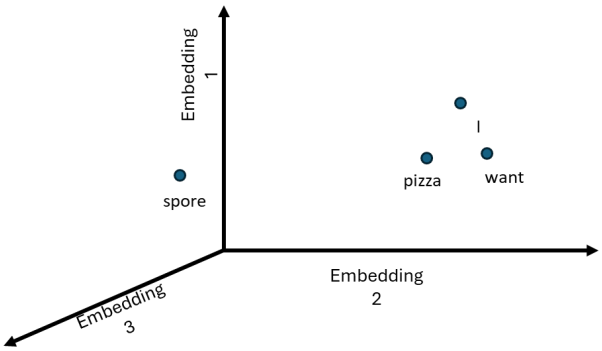


Figure 9. Word tokenization and embedding example.

5.3 Transformer

Following the embedding step, another critical component in NLP is building a network that predicts the next token probability out of the token vocabulary. Transformers (Vaswani et al., 2017) models designed to translate languages, token by token, later became the foundational structure for generative LLM (e.g., Chat GPT). To understand the transformer details, let's continue the previous example in which we want to predict the next token given "I want pizza" as input. The corresponding token embeddings in the vocabulary are stacked to form a $3 * n$ matrix E:

$$\begin{vmatrix} e_{i1} & e_{i2} & e_{i3} & \dots & e_{in} \\ e_{w1} & e_{w2} & e_{w3} & \dots & e_{wn} \\ e_{p1} & e_{p2} & e_{p3} & \dots & e_{pn} \end{vmatrix}$$

Next, we duplicate three identical matrices in which $Q = K = V = E$. Using the attention mechanism in Equation 20, the attention output (also a $3 * n$ matrix) is calculated and passed to a series of neural networks. This network will predict a list probability corresponding to each word in the vocabulary and the highest probability with the word will be selected to be the next work. For example, "from". Next, "want pizza from" will be treated as input and predict the next token and so forth.

$$attention = (QK^T)V \quad (20)$$

6. Conclusions and Future Outlook

This chapter has laid the foundational understanding of neural networks and how they power modern AI and ML applications across image processing and language understanding. Starting with an Excel-based toy example, we progressed to convolutional and transformer-based architectures.

Key takeaways include:

- Neural networks use forward passes and backpropagation to iteratively update weights based on loss gradients.
- Image data is inherently spatial, requiring CNNs for effective feature extraction.
- Language is sequential and contextual, motivating the use of transformers and self-attention.
- Loss functions, activation functions, and evaluation metrics are critical tools for training and assessment.

Moving forward, AI systems are expected to become increasingly multi-modal, combining text, image, and audio inputs. Challenges related to ethics, fairness, and explainability remain open areas of research. Educators and policymakers must work hand-in-hand with technologists to ensure AI benefits are inclusive and accountable.

Reference

Daubechies, I., DeVore, R., Foucart, S., Hanin, B., & Petrova, G. (2022). Nonlinear Approximation and (Deep) $$\mathrm {ReLU}$$Networks. *Constructive Approximation*, *55*(1), 127–172. https://doi.org/10.1007/s00365-021-09548-z

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. *Proc. IEEE Int. Conf. Comput. Vis.*, *Conference Proceedings*, 2961–2969.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, *Conference Proceedings*, 770–778.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional

neural networks. *Proceeding Adv. Neural Inf. Process. Syst.*, *Conference Proceedings*, 1097–1105.
LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.
Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *ArXiv Preprint ArXiv:1301.3781*.
Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543.
Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, *Conference Proceedings*, 779–788.
Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *Int. Conf. Med. Image Comput. Comput. Interv.*, 234–241.
Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Adv. Neural Inf. Process. Syst.*, *30*.