Unit 02 - Deep Learning (Technical Article)

1 Neural Networks

1.1 Introduction

To understand deep learning (DL), we must first look at neural networks. These are the backbone of most modern DL model architectures. So understanding neural networks, how they work, and where their 'power' comes from is integral to understanding deep learning.

Neural networks are learning algorithms loosely modeled after the brain. In the brain, there are **neurons**, which process information. Neurons are connected to other neurons through **axons**, which send information from neuron to neuron. We *heavily* abstract this system by representing neural networks as graphs. The nodes of the graph operate as neurons and the edges connecting 'neurons' act as axons. The figure below shows how a graph like this might appear.

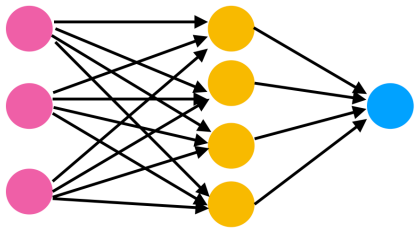


Figure 1: A visualization of neural networks

1.2 Network Structure

How let's focus on deep learning. Deep learning is a specific field of algorithms within the spheres of machine learning and artificial intelligence. As you can see in Figure 1 above, there are different levels or layers of neurons (pink, then yellow, and finally blue). That is a key characteristic of deep learning, a learning algorithm that uses **hierarchical layers to process information**.

The primary layer of neurons is called the **input layer** - this is the layer where our input is entered, as a single-column vector. So if, for example, our input was an image (which is common as deep learning algorithms are often used for image-classification), the image would be reconfigured into a large single-column vector, where each entry would represent a pixel in the image. The image would be, technically, entered into the model as one long vector of pixels, and this vector would be entered into the input layer of the model.

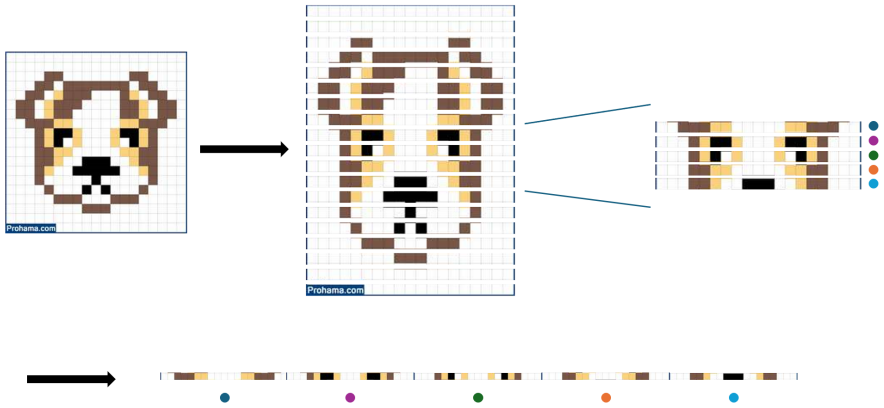


Figure 2: Visualization of how images are converted into vectors

Next are the **hidden layers**, which are yellow in the first image. This is where the actual processing is done. There are usually multiple hidden layers in deep learning models, due to how much processing the model must do before it can make a conclusion. The hidden layers are called "hidden" because we often don't

understand what happens in here. Trying to interpret the vectors that exist in these layers usually results in nonsense. Through a process called backpropagation, which we will discuss later in this article, the algorithm identifies 'features' or things it can use to classify parts of the input to make its conclusion. Referring to Figure 2, if we wanted our algorithm to classify the given image as a dog, some features might be the triangles that make up the dog's ears, the coloration of the dog, or the pink snout. The hidden layers work to identify what features are present in a given input and using these features, the computer draws a conclusion on the input.

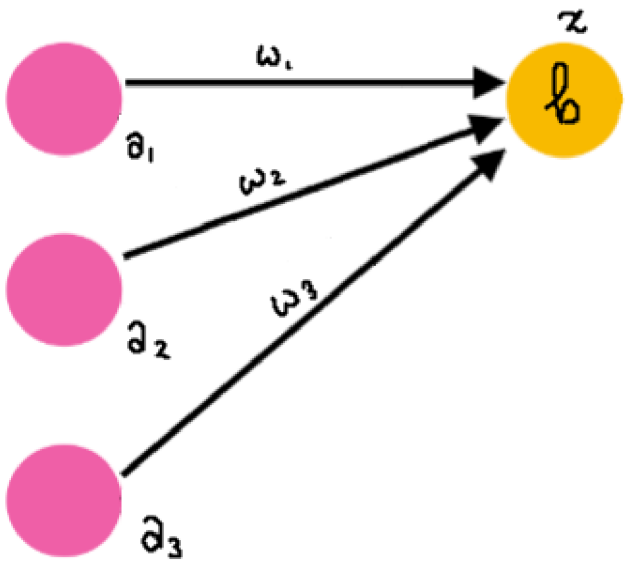
Finally, the **output layer**, which appears as blue in Figure 1. The neural network calculates a probability for each possible outcome (for example, the probability of the input image being an image of a dog) and fills out the output layer with values. The output layer shows the algorithm's conclusions for the probabilities of each possible outcome and using these probabilities, the computer chooses an outcome. For classification neural networks, all the probabilities in the outcome layer will always add up to 1. Since there is one probability calculated for each outcome, the number of nodes in the output layer will be the number of possible outcomes.

1.3 Benefits of Neural Networks

A large reason neural networks are so useful is that they don't require too much "feature engineering" to get good results. Before the use of neural network, people would write complex algorithms to extract some features from an image. These algorithms would determine the details of features like the location and size of eyes, an estimated number of legs, the length and curve of edges, etc. They would then feed these vectors into a more traditional classification algorithm like a decision tree. Neural networks are powerful enough that we can just cut up the raw image and throw it in. No complex engineering required!

1.4 Flow of Information

Now that we understand the structure of neural networks, how does information get processed and move through this structure? As we can see in Figure 1, the connections between the pink input neurons and the yellow hidden layer neurons are represented by black lines. Each of the connects are actually assigned a **weight**. These weights determine how much of the information from the previous neuron is sent to the next neuron. And the neurons themselves are also given numbers called a **bias**. A bias determines how sensitive a neuron is to input and changes in the input. Biases can be positive or negative and typically reflect the sign of the input coming into the neuron. So, for a neuron that typically receives negative data, its bias will be more positive (as in a larger positive number) to counteract that negative input.



9

Figure 3: Illustration of biases and weights in neural networks

In the figure above, a_1, a_2, and a_3 denote the labels of the three input neurons. The weight assigned to the connection between each pink neuron and the yellow neuron is denoted by w_1, w_2, and w_3 and the bias of neuron z is denoted by b. Taking in all this information,

the neuron z conducts a calculation using the formula below.

$$z = (w_1 a_1 + w_2 a_2 + w_3 a_3) + b$$

This is the value calculated by z. This value will be sent to the next neuron, which conducts a calculation using the same formula and on and on. This is how information is propagated through the neural network.

In actuality, this is a simplified version of the propagation. These calculations aren't done sequentially; rather, the network will use **matrices** to complete these calculations. The matrices will be used to compute a **dot product**, which denotes the value of z, the same value calculated by the formula above. If we take the original z equation and create a vector for the values of the neurons and a vector for the weights of each connection, we get the following formula.

$$z = \mathbf{w} \cdot \mathbf{a} + b$$

In this equation, $\mathbf{w}$ represents $< w_1, w_2, w_3 >$ and $\mathbf{a}$ represents $< a_1, a_2, a_3 >$.

Another thing we have to consider is that there are many neurons in the hidden layer, not just one. For the sake of simplicity, we narrowed our view down to one yellow hidden layer. See what happens when we include all 4 hidden neurons in this layer.

$$\begin{aligned} z_1 &= \mathbf{w}^1 \cdot \mathbf{a}^1 + b_1 \\ z_2 &= \mathbf{w}^2 \cdot \mathbf{a}^2 + b_2 \\ z_3 &= \mathbf{w}^3 \cdot \mathbf{a}^3 + b_3 \\ z_4 &= \mathbf{w}^4 \cdot \mathbf{a}^4 + b_4 \end{aligned}$$

We can simplify this function with a **weight matrix** W using matrix multiplication[1]. W holds all the weight vectors in this set of equations.

$$W = \begin{bmatrix} \mathbf{w}_1 \\ \mathbf{w}_2 \\ \mathbf{w}_3 \\ \mathbf{w}_4 \end{bmatrix} = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 \\ w_1^2 & w_2^2 & w_3^2 \\ w_1^3 & w_2^3 & w_3^3 \\ w_1^4 & w_2^4 & w_3^4 \end{bmatrix}$$

[1] If you'd like a refresher on matrix multiplication, use this website: https://www.mathsisfun.com/algebra/matrix-multiplying.html

$\mathbf{z}$ would then denote a vector holding the values of the yellow neurons in the hidden layer, as shown in the dot product calculation below. Note that in this equation, $\mathbf{a}$ is still a vector holding the values of the input neurons and $\mathbf{b}$ is a new vector holding the biases of the yellow hidden neurons.

$$\mathbf{z} = W\mathbf{a} + \mathbf{b}$$

Synthesis Questions:

1. Describe the graph representing a neural network in terms of neurobiology (use biological terms of things in the brain).
2. List the layers of a neural network and provide a bullet-point description for each layer.
3. What are weights and biases and what do they do?

2 Nonlinearity and Activation Functions

2.1 Introducing Nonlinearity

Image classification, especially with categories as specific as 'dog' or 'cat', is very challenging. It requires a level of detail and processing that many machine learning algorithms cannot achieve, especially linear algorithms, where a change in the input is directly proportional to the corresponding change in output. So, in order to tackle complex tasks such as image classification, neural networks and deep learning models need to employ **non-linearity**. In non-linear models, changes in input can cause varying levels of change in corresponding output. Going back to our dog example, if we change the size of a dog's ear, we want that to have little impact on the model's final conclusion. But if we change how the dog's fur appears, that should have a much larger impact. Non-linearity can help us achieve this. In order to introduce non-linearity to neural networking, models implement **activation functions**.

2.2 Introducing Activation Functions

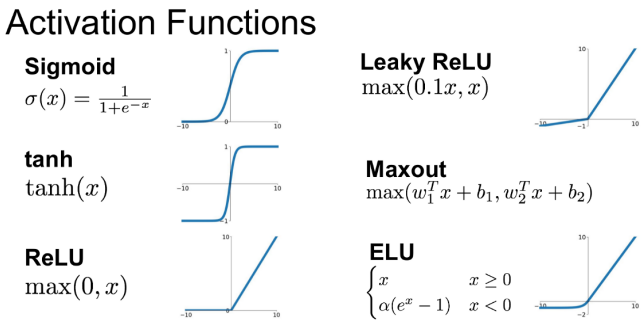


Figure 4: Plots of various activation functions

Activation functions are ways of amplifying or squashing the effect of individual neurons by assigning weights to the connections between neurons and biases to the neurons themselves. So, they can make the neurons that relate to ear size have a smaller impact and the ones relating to fur have a larger impact on the model's conclusion. These functions are applied after computing the raw values to populate the nodes of a neural network layer. These raw outputs are called **logits**. In the final z equation we found above, the values in the vector $\mathbf{z}$ would be the logits of the hidden layer.

2.3 Common Activation Functions

One commonly used activation function is the **sigmoid function**. This function squashes inputs into the range between 0 and 1. It is useful in binary classification tasks but can cause issues like vanishing gradients in deeper networks. The formula, where x is a given neuron's output is:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The hyperbolic tangent activation function, or **tanh**, squashes inputs between -1 and 1. It is zero-centered, which makes it easier for optimization compared to the sigmoid function. The tanh formula is:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Another activation function, **Rectified Linear Unit or ReLU**, outputs the input if it's positive, and zero otherwise. It is computationally efficient and helps alleviate the vanishing gradient problem by allowing gradients to flow when the input is positive. ReLU is written as:

$$ReLU(x) = MAX(0, x)$$

There is also another form of the ReLU activation function called **Leaky ReLU** that allows a small, non-zero gradient when the input is negative, helping prevent the issue of "dead neurons" (neurons that never activate). Leaky ReLU looks like:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.01x, & \text{if } x \leq 0 \end{cases}$$

2.4 Universal Approximation Theory

Using activation functions to allow for non-linearity gives deep learning and neural networking models a much greater ability to create solutions for complex tasks. In fact, a fundamental concept in the theory of neural networks, called the **Universal Approximation Theory**, states that a neural network with at least one hidden layer with a finite number of neurons can approximate any continuous function to any level of accuracy with the use of certain activation functions. So, using non-linearity, neural networks can predict almost anything and do so accurately.

Synthesis Questions:

1. Name one activation function and provide one benefit of the specific function.

3 Backpropagation

3.1 Loss Functions

How does a neural network 'learn' using training (a.k.a labeled) data? It uses a process called **backpropagation**. This process

adjusts the weights assigned to connections between neurons and the biases assigned to nodes to improve the algorithm's accuracy. Imagine for a moment, that we are working with deep neural network classifying images as dog or cat. In this case, we will want to assign a higher weight to the feature 'long tongue' because that is an important feature in determining whether something is a dog or a cat. To implement this functionality, the model will have a hidden neuron that "lights up" (has a high value) when the input animal image has a long tongue. This will significantly sway the model's decision-making process, increasing the probability of the image being a dog. In comparison, a neuron relating to the feature 'fur color' might want to have a smaller weight connecting it to the next layer, because cats and dogs have similar fur colors and this feature isn't as important in differentiating between cats and dogs. The backpropagation process will slowly adjust these weights to be this way, leading to an excellent classifier.

So how does backpropagation determine which weights and biases to manipulate? In order to understand that, we have to discuss **loss functions**. The loss function is used to measure how far off the network's predictions are from the actual target values. It provides a numeric value that represents the error in the prediction. The goal of backpropagation is to reduce this loss by adjusting the weights in the network. A common example of a loss function is **Mean Squared Error (MSE)**, which is often used for regression tasks. The formula for MSE is:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where L is the loss, or the model's error, y_i is the target-value (also known as the correct answer or label) for the specific input, $\hat{y}_i$ is the value the model predicted/guessed, and N is the number of training examples shown to the model. You may notice that this version of the MSE loss function works only with scalar outputs from a neural network. There are versions of MSE that can handle multiclass output. However, it is generally accepted that if you have a multiclass classification problem, a better loss function to use is **Cross-Entropy Loss** (or log loss). This loss function measures the difference between two probability distributions. The

formula for Cross-Entropy Loss is:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \cdot log(\hat{y}_{ik})$$

where k is one of the outcomes or classes, y_i is the actual probability of the given input belonging in class k, and $\hat{y}_i$ is the probability the model has assigned to class k for the given input.

Calculating these loss metrics is the final step of what is called the **forward pass**. This is what we have studied so far. After this begins the **backward pass** or **backpropagation**.

3.2 Optimizers

Now taking these loss functions, backpropagation manipulates different weights and biases to reduce the loss calculated, thereby increasing the accuracy of the deep learning model. To reduce the calculated loss, an optimizer, usually **gradient descent**, is employed. The first step of gradient descent is the forward pass, where the input is propagated forward through the model's layers and an output is generated. Then the loss functions are used to determine how far off those predictions are. Next, the model moves backwards through the layers and computes the gradients (or derivatives) of the loss function with respect to each weight in the network.

As you likely know, derivates show the direction in which the graph is moving at a specific point. So, in this case, taking the derivative of the loss function will show the diagonal of the function - whether the loss function value (the error of the model) is increasing as you increase the different weights in the function or decreasing. Using these gradients, we can determine how much each weight should be adjusted to reduce the loss, whether a weight should be increased, decreased, or remain as is. The formula is:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w}$$

where $\frac{\partial L}{\partial \omega}$ is the gradient of the loss with respect to weight w, $\frac{\partial L}{\partial \hat{y}}$ is the gradient of the loss with respect to the predicted output, $\frac{\partial \hat{y}}{\partial z}$ is

the gradient of the output with respect to the weighted sum z, and $\frac{\partial z}{\partial w}$ is the gradient of the weighted sum with respect to the weight w.

While gradient descent is common and easy to follow, it is very expensive computationally and can be slow. **Stochastic Gradient Descent (SGD)** is another optimizer and it speeds up this process by updating weights after computing the gradient on a small, random batch of data, rather than the entire dataset. Another popular optimizer is **Adam**. Adam combines the benefits of both SGD and another optimization technique called Momentum, which helps the optimizer move faster by incorporating information from past gradients. Adam also adapts the learning rate for each weight based on how the gradients change, making it efficient for handling noisy gradients and sparse data. Many people use Adam because it performs well across a wide range of tasks, but it's essential to understand that it fine-tunes the learning process dynamically, making it more flexible than standard SGD.

A common thread between all of these optimizers is that they share one parameter in common: the **learning rate**. This is a scaling factor applied to the calculated gradient before it is used to adjust the weights. The size of this scaling determines how big of a "step" the weights take while traversing the "loss landscape". Optimizer classes in most deep learning libraries (e.g. PyTorch, Tensorflow) have defaults that work well for each of the different optimizers. Starting here is usually a good idea, because a learning rate that is too small will take way too long to converge, while a large learning rate may not converge at all! For an illustration of this, see Figure 5.

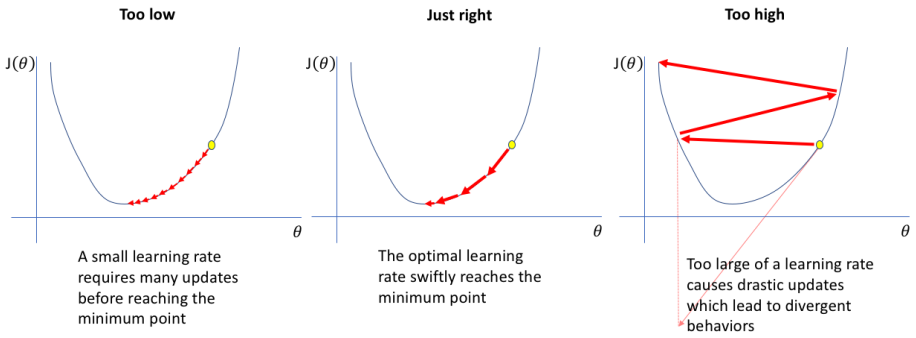


Figure 5: A demonstration of how different learning rates affect convergence of a loss function (in this graphic represented as $J(\theta)$.

Once these gradients are calculated, they are used to shift each weight accordingly and the model is run again with the new weights and biases. This is a forward pass, thus the process of backpropagation starts again.

Synthesis Questions:

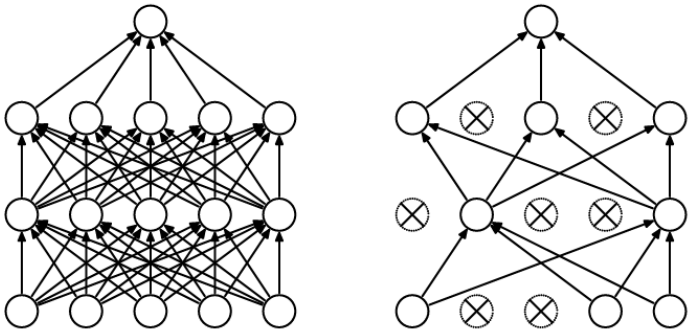
1. In backpropagation, which values are shifted to decrease the value calculated by the loss function (a.k.a. make the model more accurate)?
2. In your own words, describe why derivates of the loss function are calculated in gradient descent?
3. Why is the learning rate so important? What happens if the learning rate is too low? What happens when it is too big?

4 Regularization

We will now cover an incredibly important topic for deep learning: **regularization**. In ML, regularization often is applied in the form of adding a norm to the loss function, encouraging weights to reach smooth (L2) or sparse (L1) optima. Regularization also exists in

deep learning to prevent overfitting, but comes about in more interesting and varied ways. We will quickly cover two common ones in shallow detail.

Dropout: During each forward pass, a certain fraction of neurons are temporarily dropped from the neural network. The connections they have to other neurons are totally ignored and they are not used in the forward nor the backward pass for that specific training example. In other words, dropped neurons have no direct bearing on the output and their associated weights will not change from that training example. This only occurs at train time. At test time, all neurons are allowed to be used by the network all the time. See Figure 6 for a visual representation of this process.



(a) Standard Neural Net (b) After applying dropout.

Figure 6: Visualizing dropout in a fully connected neural network

Why would we do this? Well, if a network is allowed to use all of its neurons all of the time, there is no guarantee that the network will *effectively* use all the neurons. Perhaps due to a strange initial weight configuration, the random seed, or some other factor, the network will only use a very small subset of its available power. Many neurons will end up going essentially unused and their weights meaningless, despite the resources being allotted to train them. This consolidation of computation into a small portion of the network leads to much lower robustness.

Here is an analogy: a 10-armed robot is trained to pick up an

apple from a desk. The robot uses one of its arms at random, and only trains with that arm. Then during test time, we bind the arm the robot used the most. Since the robot has not learned how to use any of its other arms, it fails at a task it should very easily be able to do. Dropout is like forcing the robot to pretend it has lost a few arms each training example. It is then forced to use all of its arms, and gets good at picking up the apple with any of them. This robot is now much more robust when deployed into the real world, as small perturbations do not totally handicap it.

One small detail is that once training time is over, all neurons have their weights scaled by 1 minus the dropout rate. Therefore, the expected distribution of values flowing from one layer to the next stays the same as it was during training time.

Batchnorm: Batchnorm, or batch normalization, is a very common technique used to regularize neural networks and improve training efficiency. Batchnorm can be thought of as an extra layer in a neural network with a few additional parameters. Essentially, before information flows from one layer to the next, the data is **normalized** across examples in the batch. Let's break this down.

What is a **batch**? It is common to not just pass one example through a neural network at a time, but many. Groups of 64, 1024, 4096, etc. training examples are used at once before a backward pass is performed. Then what is **normalization**? Within this aforementioned batch, examples are mean-shifted by a mean μ and then divided by a standard deviation σ. This is normalization and it keeps the data centered around $(0, 0)$ and evenly varied. In addition, these μ and σ parameters are slightly adjusted for each new batch by taking a moving average through each previous batch. There are also additional "scale and shift" parameters represented by γ and β respectively. As the network sees more and more examples, the β and γ parameters are also slowly adjusted to find a scale and shift transformation to the normalized data that improves performance. This is shown in the following series of equations, where x_i is a datapoint and m is the number of datapoints in a batch.

Estimating μ and σ for a batch:

$$\mu_{batch} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{batch} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{batch})}$$

Updating the moving average:

$$\mu = \alpha\mu + (1-\alpha)\mu_{batch}$$

$$\sigma = \alpha\sigma + (1-\alpha)\sigma_{batch}$$

Normalize x_i to $\tilde{x}_i$ using the moving average:

$$\tilde{x}_i = \frac{x_i - \mu}{\sigma}$$

Scale and shift before sending values to next layer:

$$x_i^{output} = \gamma\tilde{x}_i + \beta$$

Mean-centered and evenly varied data allows gradient descent to descend the weights smoothly down to a optimal minimum. If two features input into a neural network were on wildly different scales (e.g. House price vs. number of bedrooms), then optimization becomes clunky. Making a large shift in weights is necessary to process highly varied housing prices, but this same shift negatively affects how the network processes the more tame "number of bedrooms" feature since more precise changes are needed. Normalization plus scale and shift removes this problem, and thus batchnorm helps greatly with a network's convergence.

Synthesis Questions:

1. In your own words, why does Dropout work?
2. Think of, or find online, a method of regularization within neural networks not discussed here. Write two to three sentences on how it works.

5 When to Use Neural Networks

While we have shown the great power and ability of neural networks and deep learning, with all this power comes great cost. Neural networks are power-hungry, taking up significant computation power for all the processing they have to do. They are a powerful tool and should be used with caution and respect. As our i2 president once said, **don't use a bomb to cut a sandwich**. It is important to know when to use them. Deep learning models are useful for image, audio, video classification or in other cases where non-linearity is necessary. Neural networks also require considerable amounts of data for a decent accuracy. Only use them when you have enough labeled data to properly train them on. In summary, while neural networks offer unmatched capabilities for complex tasks like image and audio classification, they are not always the most efficient tool. Sometimes a simpler model will get the job done while using half the resources. The key is understanding when their power is necessary and when a more lightweight solution will suffice. Always choose the right tool for the job.

Technical Project Spec

The project for this "Basics" section will have you **finish a code template through Google Colab**. Please ask questions as you work through this project. Be sure to discuss with others in your group if you have one! Share your answers as you like, the goal is to learn and we're not holding grades over your head.
In this project, you will be implementing a Deep Neural Network (DNN)!

A few general helpful tips (if applicable):

- Be sure to appropriately make a copy of the Colab template before starting to save your progress!
- Renaming your copy to something that contains your name is a good idea, it will make it easier for us to review your submissions.

- Leave comments to cement your understanding. Link syntax to ideas.
- Read up on what **MNIST** is.

Now, follow the instructions on this Jupyter notebook to implement some of the things we talked about. There is an "answers" link at the bottom of the notebook that you can use if stuck. You will need to download the '.ipynb' found in that directory and open it either locally or in a new colab project yourself. Ask around if you are unable to get it working!

Colab Link: Unit 02 Notebook (1 hr)

When you are finished with your code, independently verify that it works and have fun with it! If you add any additional functionality be sure to talk about it with others and give them ideas.

Remember that this is all for your learning, so do your best and don't stress!

Congratulations! You now understand the basics of Deep Neural Network structure, how they learn, and how to create one using Python!