Exploration of Artificial Intelligence as a Career Field

Coby L. Kassner
[redacted]

MAT 2420-C04 Calculus II : MA1

July 31, 2023

1 Introduction

For this project, I explored the career field of software engineering—specifically, the careers surrounding artificial intelligence and machine learning. Artificial intelligence is part of a broader field of study that includes logic, search, optimization, probabilistic reasoning, and machine learning—which generally utilizes various architectures (e.g., neural networks) to solve data-oriented problems that cannot easily be solved with any straightforward algorithm.

Artificial intelligence architects often spend time developing and implementing AI systems in various new use cases. Day-to-day work includes lots of programming and debugging, but also many meetings, check-ins, and brainstorming sessions, as well as time spent researching libraries or new potentially useful implementations. In more research-oriented projects, developers can be at the absolute bleeding edge of computer science thought, writing papers that may push the entire industry forward (e.g., Vaswani et al., Kingma and Ba, etc.). Perhaps the most significant and fascinating problem in the field of artificial intelligence is the concept of artificial general intelligence (AGI). Theoretically, an AGI would be an architecture with the capacity to outperform humans at practically any task. If an AGI reached a certain threshold of intelligence, it could be able to create a more intelligent version of itself. Accordingly, this even more intelligent version could program a better version of itself, and self-replication could continue until the agent reached some physical or practical limit to intelligence. The event of an agent reaching such a threshold is known as "the singularity"—see Appendix 1 for a more nuanced discussion.

Typically, a career in artificial intelligence requires a Bachelor's degree in Computer Science. Graduate degrees are not usually required, but certain jobs may require a Master's degree or a doctorate. As a whole, AI is a rapidly-growing career field with the potential to revolutionize humanity's relationship with modern technology.

2 Interview

For this project, I interviewed [redacted], an Artificial Intelligence/Machine Learning Architect at [redacted]. [redacted] has over 25 years of experience in the Defense and Aerospace Industries and has worked as a software engineer for companies of all sizes (everywhere from LLCs with less than 30 employees to Lockheed Martin with over 100,000 employees). [redacted] has a Bachelor's degree in Electrical and Computer Engineering and a Master's degree in Computer Science.

Here are the takeaways from our interview that stood out to me the most:

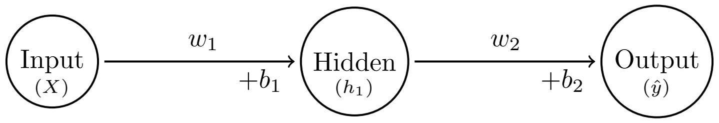
1. Workdays are highly dynamic, especially with many employers still offering remote or hybrid positions. Every day differs, especially regarding the type of work being done.
2. As difficult as the mathematical and technical aspects of computer science may seem, anyone with "a decently rational mind" can do them—but to have a genuinely successful career, soft skills cannot be disregarded. Particularly, the ability to communicate technical information clearly and understandably is a desirable skill, as well as the ability to collaborate and build rapport with colleagues.
3. Math undergirds the entire field of computer science—even the binary logic at the most basic levels of processing is math. However—practically—most of the math (e.g., matrix calculus algorithms) is buried deep inside software libraries. Knowing how to utilize and implement software does not entail an in-depth understanding of the underlying algorithms (that is, of course, unless you are designing the algorithms in the first place).

[redacted] also said that [redacted] favorite part of [redacted] job was working with [redacted] team, and [redacted] least favorite part was the restriction on creativity that comes from working from home.

3 Math Problem

For my math problem, I will train and test a single hidden layer, one-dimensional (1 input, one hidden neuron, and one output) feed-forward neural network by hand. This is precisely the type of math that would be left to an algorithm in a software library, but I could not think of any other more practical problems that distinctly involved calculus. I will be simplifying and glossing over a lot of things here—see Appendix 2 for a short background on neural networks, if needed.

Here is the architecture of our neural network:



w_1 and w_2 are our weights, and b_1 and b_2 are our biases. propagating forward (getting output from input) through this network is quite simple—we multiply our input by w_1 and add b_1 to get the activation of our hidden neuron. Then, we multiply this by w_2 and add b_2 to get our output. In larger networks, there are many more weights and neurons to account for, so activation is calculated through matrix multiplication. Anyway, I want to train our network to output -1 when we feed in 1, and 5 when we feed in 2 (we really don't need a hidden neuron for this, but with just input and output, what we'd be doing would not really be backpropagation). These will be the only data points we feed the network. This value that we want the network to output is called the label and is notated as y. We'll initialize our weights and biases to 1 for simplicity, though they are usually randomized. Also, we'll calculate our loss function. This is essentially a measure of how "bad" our output is compared to the label.

Let's forward propagate:

1. $X = 1$; $y = -1$; $h_1 = X \cdot w_1 + b_1 = 2$; $\hat{y} = h_1 \cdot w_2 + b_2 = 3$; Loss $(L) = (y - \hat{y})^2 = 16$
2. $X = 2$; $y = 5$; $h_1 = X \cdot w_1 + b_1 = 3$; $\hat{y} = h_1 \cdot w_2 + b_2 = 4$; Loss $(L) = (y - \hat{y})^2 = 1$

Now, in order to get our desired outputs, we have to calculate the gradient of the loss function—we have to figure out the multi-dimensional "slope" of the function so that we can move "down." In other words, we have to figure out how much and in what direction we should change the weights and biases to minimize our loss and thus achieve our desired output. First, we want to find the partial derivative of the loss with respect to w_2 by finding these partial derivatives (we can use the chain rule!):

$$\begin{aligned}
\frac{\partial L}{\partial w_2} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} \\
L &= (y - \hat{y})^2 \\
\frac{\partial L}{\partial \hat{y}} &= 2(y - \hat{y}) \\
\hat{y} &= h_1 \cdot w_2 + b_2 \\
\frac{\partial \hat{y}}{w_2} &= h_1 \\
\therefore \frac{\partial L}{\partial w_2} &= h_1 \cdot 2(y - \hat{y})
\end{aligned}$$

This tells us how to change w_2 to minimize our loss function, and we can find the derivative of the loss with respect to b_2 from this quite easily as well. Often, biases are conceptualized as if they are an additional, separate thing that you add to each neuron. However, this is a weird and often cumbersome way of thinking about them. Instead, we can see our biases as weights connected to special neurons that always have an activation of 1. This way, instead of having to wade through several more partial derivatives to get to the same answer, we can copy our equation for the partial derivative of the loss with respect to w_2 and just replace h_1 with 1:

$$\frac{\partial L}{\partial b_2} = 2(y - \hat{y})$$

Next, we can find the partial derivative of the loss with respect to h_1:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1}$$
$$\hat{y} = h_1 \cdot w_2 + b_2$$
$$\frac{\partial \hat{y}}{\partial h_1} = w_2$$
$$\frac{\partial L}{\partial h_1} = 2(y - \hat{y}) \cdot w_2$$

Now, this is where the idea of backward propagation comes in—we can find the rest of our partial derivatives very easily now. Since the partial derivatives of h_1 with respect to w_1 and b_1 are algebraically identical to the partial derivatives of $\hat{y}$ with respect to w_2 and b_2, we don't need to do any more math to calculate those, and we can simply multiply those partials by the derivative of the loss with respect to h_1 to find the derivative of the loss with respect to w_1 and b_1!

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial w_1}$$
$$\frac{\partial L}{\partial w_1} = 2(y - \hat{y}) \cdot w_2 \cdot X$$
$$\frac{\partial L}{\partial b_1} = 2(y - \hat{y}) \cdot w_2$$

Let's find these for the first training example ($X = 1$; $y = -1$):
$\frac{\partial L}{\partial w_2} = -16$; $\frac{\partial L}{\partial b_2} = -8$; $\frac{\partial L}{\partial h_1} = -8$; $\frac{\partial L}{\partial w_1} = -8$; $\frac{\partial L}{\partial b_1} = -8$.

Let's find these for the second training example ($X = 2$; $y = 5$):
$\frac{\partial L}{\partial w_2} = 6$; $\frac{\partial L}{\partial b_2} = 2$; $\frac{\partial L}{\partial h_1} = 2$; $\frac{\partial L}{\partial w_1} = 4$; $\frac{\partial L}{\partial b_1} = 2$.

Now, we can average them:
$\frac{\partial L}{\partial w_2} = -5$; $\frac{\partial L}{\partial b_2} = -3$; $\frac{\partial L}{\partial h_1} = -3$; $\frac{\partial L}{\partial w_1} = -2$; $\frac{\partial L}{\partial b_1} = -3$.

Now, theoretically, we are done! All we need to do is adjust our weights and biases by these partial derivatives, and we should have the correct numbers, right? Unfortunately, things are not so simple. There are several more factors we need to consider, the most significant of which is that the gradient at this point does not "point" exactly at the local minimum we are trying to reach. Thus, if we just adjust our weights by these partials, then we will miss or overshoot our minimum, and we won't be in any better of a position than when we began. Instead, we have to take baby steps, slowly moving towards our minimum and re-calculating at every step. This process is called gradient descent. We take small steps by multiplying these partials by some constant η ($\eta < 1$), called the learning rate, to lessen their impact, so to speak. Again, after adjusting the weights and biases, we re-calculate our partials, then re-adjust, and repeat the process.

After 1,000 such steps, with a learning rate of $\eta = 0.01$, the weights and biases are the following:
$w_2 = 2.79342$; $b_2 = -1.25574$; $w_1 = 2.14791$; $b_1 = -2.05636$.

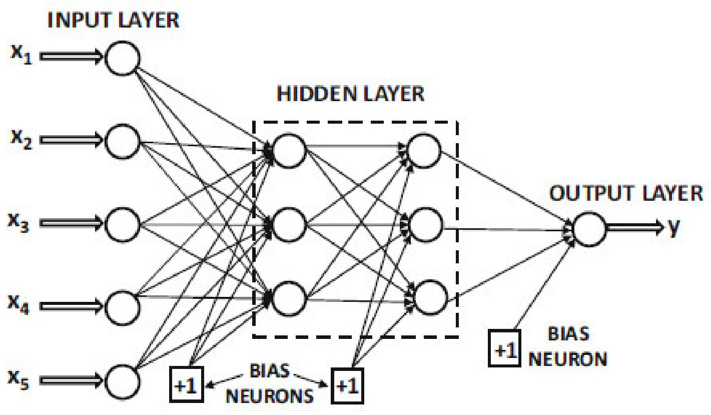
These output -1 when inputted 1, and 5 when inputted 2. The loss is effectively infinitesimal.
I was very reluctant to do this, but after multiple attempts backpropagating around 10 steps by hand with a high learning rate and having the weights diverge several times, I realized that, even if I were able to do all of the calculations by hand, there would be no real advantage to it—I already did the important part of the math (deriving the formulas; everything left is just plugging numbers in)—and it would probably take up several pages, which is space that I don't have. Because of this, I wrote a very simple script in Python to do the raw calculations—find it in Appendix 3. This was a very effective reminder of exactly why backpropagation calculations are left to computers.

Appendix 1

While many argue that such a singularity is possible—or even inevitable, I find that there are some fundamental issues with this notion that need to be addressed before any rigorous discussion can take place. Particularly, I would argue that we do not have any real, comprehensive definition of intelligence (Shevlin et al.), and the idea that intelligence—whatever it may be—enables the creation of more intelligence makes this significantly more problematic. Somewhere in the Library of Babel, there is the Python code written for an AGI with capabilities beyond our wildest imagination. How smart is the primate inside the infinite monkey theorem, then, considering that it will program the same AGI at some point? What if there was an AGI that surpassed humans at every task except programming? What about the inverse?

Appendix 2

Neural networks are a very complicated subject. In short, they are a very loose analogy for how our brains function. There are layers of neurons. Each neuron is connected to other neurons with connections called weights. Weights multiply their input. There are also biases, which offset the neurons to which they are connected. The activation of any given neuron is given by the sum of the connections from the previous layer through the weights, and a bias. Often, there is also an activation function through which the activation is fed to get a final activation (e.g., Sigmoid, which squeezes the activation to between 0 and 1), though this is one of the things I ignored in the problem. You can *forward propagate* through a neural network by calculating all of the activations of the neurons—this gets you output from input. You can *backpropagate* through a neural network by using the backpropagation algorithm, which uses gradient descent to incrementally adjust the weights and biases until the network approximates the function from input to output of your data with sufficient accuracy. Backpropagation "trains" your neural network to perform better. Neural networks are universal function approximators, meaning that a sufficiently large neural network could, theoretically, approximate any conceivable, consistent function.



If you are unfamiliar with neural networks, I would highly recommend watching this video series by 3Blue1Brown, which explains neural networks far more intuitively and in more depth than the above summary. If you want an extremely in-depth, graduate-level overview of neural networks, I can email you the textbook from which I am self-studying (Neural Networks and Deep Learning by Charu C. Aggarwal).

Appendix 3

```python

w_1 = 1
w_2 = 1
b_1 = 1
b_2 = 1
data = [[1, -1], [2, 5]]


def forward_propagate(data_point: list[float]) -> float:
    return w_2 * ((w_1 * data_point[0]) + b_1) + b_2


def get_loss(data_point: list[float]) -> float:
    y_hat = forward_propagate(data_point)
    y = data_point[1]
    return (y - y_hat) ** 2


def get_partial_Lw2(data_point: list[float]) -> float:
    x = data_point[0]
    y = data_point[1]
    y_hat = forward_propagate(data_point)
    h_1 = x * w_1 + b_1
    return h_1 * 2 * (y - y_hat)


def get_partial_Lb2(data_point: list[float]) -> float:
    y = data_point[1]
    y_hat = forward_propagate(data_point)
    return 2 * (y - y_hat)


def get_partial_Lw1(data_point: list[float]) -> float:
    x = data_point[0]
    y = data_point[1]
    y_hat = forward_propagate(data_point)
    return x * w_2 * 2 * (y - y_hat)


def get_partial_Lb1(data_point: list[float]) -> float:
    x = data_point[0]
    y = data_point[1]
    y_hat = forward_propagate(data_point)
    return w_2 * 2 * (y - y_hat)


for i in range(1000):
    partial_Lw2 = 0.01 * (get_partial_Lw2(data[0]) + get_partial_Lw2(data[1]))
    partial_Lb2 = 0.01 * (get_partial_Lb2(data[0]) + get_partial_Lb2(data[1]))
    partial_Lw1 = 0.01 * (get_partial_Lw1(data[0]) + get_partial_Lw1(data[1]))
    partial_Lb1 = 0.01 * (get_partial_Lb1(data[0]) + get_partial_Lb1(data[1]))

    w_2 += partial_Lw2
    b_2 += partial_Lb2
    w_1 += partial_Lw1
    b_1 += partial_Lb1

print(
    f"w_1: {round(w_1, 5)}, b_1: {round(b_1, 5)}, w_2: {round(w_2, 5)}, b_2: {round(b_2, 5)}"
)
print(f"Total loss for all data: {get_loss(data[0]) + get_loss(data[1])}")
print(f"Input: 1, Output: {round(forward_propagate(data[0]), 5)}")
print(f"Input: 2, Output: {round(forward_propagate(data[1]), 5)}")

```

Output:

```

w_1: 2.14791, b_1: -2.05636, w_2: 2.79342, b_2: -1.25574
Total loss for all data: 6.310887241768094e-29
Input: 1, Output: -1.0
Input: 2, Output: 5.0

```

Works Cited

Aggarwal, Charu C. Neural Networks and Deep Learning : A Textbook. Springer International Publishing, 2018.

Copeland, B.J.. "artificial intelligence". Encyclopedia Britannica, 25 Jul. 2023, https://www.britannica.com/technology/arti intelligence.

Kingma, Diederik P., and Jimmy Ba. "Adam: A Method for Stochastic Optimization." ArXiv.org, 22 Dec. 2014, arxiv.org/abs/1412.6980.

Shevlin, Henry, et al. "The Limits of Machine Intelligence." EMBO Reports, vol. 20, no. 10, Sept. 2019, https://doi.org/10.15252/embr.201949177.

Vaswani, Ashish, et al. "Attention Is All You Need." ArXiv.org, 2017, arxiv.org/abs/1706.03762.