Research Journal

Shad Boswell
University of Utah

2023

June

Jun. 11, 2023 — Sunday

My research today was split between reading d2l.ai (Smola n.d.) and watching Andrej Katpathy's video Karpathy n.d.(b)

Notes on backpropogation and neural networks:

1. Neural networks are just a mathematical expression which can be drawn out using a tree. This tree helps to visualize the expression and understand how backpropogation works
2. Backpropogation involves starting with the result of an expression and working backwards to see how each parameter to the equation influences the total outcome. It's a "recursive application of backpropogation through the computational tree"
3. A loss function is a way to tell how well your model is performing. In machine learning, we are trying to minimize our loss as much as possible.
4. Neurons in a neural network take all the summation of the weights * inputs, add a bias and pass the result to what is known as an "activation function" or "squashing function".
5. Squashing functions are typically a sigmoid or tanh function and map the real numbers to the range [0,1]

Jun. 12, 2023 — Monday

Finishing Andrej Karpathy's video (Karpathy n.d.(b)). Notes:

1. Practice backpropogation. Rules for backpropogation:
    a) Backpropgation through addition disperses the gradient to all of the child nodes

   b) Backpropogation through multiplication takes the the other children's data, multiplies them, and then multiplies by the output gradient
   c) Generally, we take the local derivative of the function and multiply by the output gradient

2. Learn the subtleties of Python. The "__rmul__" function serves as fallback for the "__mul__" function and swaps the order of the operands

3. Practice using things that you have already written. Example: Andrej implemented subtraction as the addition of a negation. Andrej had already written the addition operation so he just need to write the negation operation which is just a multiplication by -1. Andrej had already written multiplication so this was trivial.

4. Pararphrase: "The level at which you implement your operations is up to you... All that matters is that we have some kind of inputs and some kind of outputs and that the output is a function of the input... As long as we can forward pass and backward pass, then the design is up to you." - Karpathy n.d.(b)

5. Using PyTorch and Tensors like Micrograd:
   a) Need to specify that tensors need a gradient with: n.requires_grad = True
   b) Tensors have .data.item() which strips out the value in the Tensor
   c) Tensors also have .grad.item() attribute similar to Value.grad in micrograd
   d) The big deal with PyTorch is the efficiency and our ability to take advantage of parallelism with Tensors

6. Building a Nerual Net. A neural net is a specific class of mathematical expression.
   a) We start by building a Neuron class and then initializing it with some weights and a bias.

7. Loss is a single number that represents the overall performance of our model. In essence it is the difference between the actual output values and the predicted values. Thus, we want to minimize our loss so that our model performs how we expect and accomplishes the task at hand.

8. Common loss function is the mean-squared-error function:

$$\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

   where Y_i is the target values and $\hat{Y}_i$ is the observed values. Why MSE? The further we are away, the greater the loss would be. This effect is exaggerated by squaring the difference between the values. Any difference less than one will also result in a smaller loss than difference which rewards small loss. In a perfect world, our target and our observed value would be the same which would yield 0 loss.

9. Do not mess with inputs to neural nets only weights and biases. The total number of parameters for our neural net is the total number of weights + the total number of biases. We can find this by taking the sum of products between layers (weights) and adding the number of neurons in the inner layers + the outer layer (biases)

10. Modifying parameters:

    a) If a gradient is negative, you increase the parameter

    b) If a gradient is positive, you decrease the parameter

    c) We can simply just multiply our step size by the -gradient

    d) Andrej states that our gradient vectors point in the direction of increasing loss, so we want to go in the opposite direction.

    e) Then you just iterate this process until loss is really low (low enough for whatever your standards are). This is called **gradient descent**!

    f) Be careful, you may overstep! Similar to Newton's method -> You will bounce back and forth between having close observed values and far away observed values. This is called learning rate. We need to find the step size to be just right. You can either take way too long, or explode your loss.

11. Training loops:

    a) First do forward pass, then backward pass so we have accurate gradients. Then update. This is the fundamental idea of gradient descent!

    b) **MAKE SURE YOU FLUSH THE GRADIENT BEFORE BACKWARD PASS!!**

    c) Learning rate decay: Slowing down the learning rate with more iterations (Scaling the learning rate as a function of the number of iterations) allows you to focus on the finer details as the network learns more and more.

12. Summary:

    a) Neural nets: Mathematical expression for a forward pass, followed by a loss function. We then do a backward pass to get the gradient so we can tune each of the parameters to decrease the loss locally.

    b) When the loss is low, the network is performing how we would like.

    c) ChatGPT essentially takes a series of words and attempts to predict the next word in the sequence. When you train this on the internet, the network has really amazing emergent properties and has billions of parameters.

Reading d2l.ai staring at section 3.1: Regression
Meeting with Ganesh

1. You need to share Tensors. That is the name of the game with machine learning.

2. groq inc paper - Software-defined tensor streaming multiprocessor for large-scale machine learning. They have their own chips and they essentially want to get rid of the idea of kernels.

3. NaN in the GPU: Do not have a great status. In the CPU you can use a specific bit pattern to quiet the NaN or make it signal. Read "The Secret life of NaN"

4. Attacks to floating point computation

    a) Injection exceptions in the input to see how far it goes
    b) Lowering precision for some stages
    c) Alterting the inputs to consume a larger dynamic range
    d) Changing hyper parameters
    e) Changing how the libraries are sparsified

    How do we find and locate these vulnerabilities? What is the least number of monitoring layers to ensure security of the network?

5. Look at FPTuner

Watching Andrej Karpathy's video Karpathy n.d.(a)

1. Makemore is designed to makemore of whatever you feed it.

2. Makemore is a character level language model. It treats every single line of input as an example and then each as sequences of characters. This is the level on which we are building out makemore.

3. With character level language models, we need to be careful of all of the examples packed into a single word. For example, consider the word "isabella". The examples embedded in this word are:

    a) "i" will come first
    b) "s" is likely to follow an "i"
    c) "a" is likely to follow "is"
    d) and so on...
    e) until we get to the end where the word where "a" is likely to follow "isabell"
    f) There is one last example and that is that the word is likely to end after "isabella"
    g) With long words, we have a **TON** of information

4. A bigram is where we try to predict the next letter by only looking at the current letter.

5. The zip function in python is great! It takes two iterators together and pairs the iterations together, truncating the longer iterator to meet the shorter one.

6. What are the statistics involved in a bigram? The simplest way is just through counting.

    a) Let's keep a dictionary to keep track of the counts of tuples. How? Make a tuple of the characters, get its value from the dictionary (default to 0) and add 1 to it. Then restore it.

    b) It's actually going to be more convenient to store these counts in 2d arrays where the rows are the first character in the bigram and the columns are the second character in the bigram. We will use PyTorch's Tensor

    c) Tip: Use setbuilder notation to build lookup tables!

    d) Tip 2: plt.imshow() is pretty cool for plotting 2d arrays

7. Now that we have the counts, we can just sample our data repeatedly to create a new word. We can do this by creating probabilities for the starting characters. This is done by normalizing the array of starting characters.

8. We then use Python's multinomial function to feed it a probability distribution which it will use to pick a number. The numbers than it can select from are the same as the indices of the distribution. Andrej also used a Python generator object to provide a dterministic way of sampling the probability distribution. (I think was for teaching purposes only and maybe not advised? I will have to look into generators even more, but I think they always pick the same if you give them a specific seed.)

9. All in all, bigram language models are pretty terrible. However, they still do something reasonable. If we compare the results to a model that is completely untrained, it performs far better. The words are actually word like where the untrained model just spits out garbage.

10. Once you are finished writing a model, be sure to look for inefficiencies that can be solved easily. (Remember Kopta: "The #1 rule of optimization is: Don't"). The optimization that Andrej did was to factor out the probability calculation from the loop. Rather, we would like to create a 27x27 matrix of probabilities that we can read from. Ideally, we would like each rows probability to be calculated in parallel.

11. We can also sum along dimensions with Tensors. For example, with a 2d Tensor, summing along dimension, or axis, 0 will sum the values in the columns to create a 1 x N Tensor. On the other hand, summing along dimension, or axis, 1 will sum the values in a row to create an N X 1 Tensor. This is really helpful because we can then take advantage of Broadcasting! Wooo!

12. Andrej's tips: READ THROUGH BROADCASTING SEMANTICS! Treat it with respect and take your time.
13. In place operations have the potential to be faster. Use them when you can.

Jun. 13, 2023 — Tuesday

I spent most of the day reading from d2l.ai Chapter 3. See the "d2l.ai Notes" Overleaf for updated notes.

Andrej Karpahty Video: Karpathy n.d.(a)

1. Likelihood estimation. Your model assigns probabilities to every prediction that it makes. If you take your training set and look at the probabilities for each step, they should be close to 1 if your model is performing well. The higher these probabilities are, the better that your model is performing. This is because it is accurately predicting what it needs to predict or what it was trained to predict. The likelihood of a model is the all of the probabilities in your training set multiplied together. This gives an indication of how well your model can predict the entire data set. Since these are less than 0, you often end up with a really small number. Thus, most people work with the log of the likelihood. The log of the likelihood approaches 0 as your likelihood approaches 1 and $-\inf$ as your likelihood approaches 0. This is mostly for convenience: $log(a * b * c) = log(a) + log(b) + log(c)$
2. How high can log likelihood get? 0 when all probs are 1, $-\infty$ when all probs are 0. This is counter-intuitive with respect to the loss function so we will work with the negative log likelihood instead. This just flips the log around the x-axis. Thus, the lowest you can get is 0 which is the most accurate and the highest you can get is ∞ which is the least accurate.
3. We want to find the parameters that minimize the negative log likelihood (maximize the likelihood). The average negative log likelihood summarizes your model.
4. Model Smoothing: Add some fake counts to everything to smooth out the model. We do not necessarily want to get rid of every single probability. As long as we keep the model smoothing small, our model will still think that the things that were previously 0 are unlikely but will not return an infinite loss.
5. Now lets create a neural network for our bigram language model. Pretty much the same approach but with weights, biases, and backpropogation through our loss function. Steps:
    a) Create training set of bigrams. This will be made up of two lists: the inputs and labels (targets). Both will be integers to our lookup table. We will then create tensors from these. Example: "emma" has 5 examples for our model: [.e, em, mm, ma, a.]. Thus, our training data will be: [0, 5, 13, 13, 1] and [5, 13, 13, 1, 0]

b) Tangent: Use torch.tensor not torch.Tensor. torch.Tensor will set dtype to float32 no matter what the dtype actually is. torch.tensor will infer the dtype from what you feed in.

c) Tangent: You cannot just plug an integer index into a neural net. We need to pass in a vector. (It really would not make sense to pass an integer index into a neural net. This would skew the weights and the activation of the receiving neuron.) Thus, we introduce one hot encoding. This is a common way of encoding integers into a neural net. Example: For the integer 13, we create a 1X13 vector of all 0's besides the 13th one is a 1. When we plug numbers into neural nets, we want them to be floating type so they can take on a range of values.

d) Initialize the weights using randn. We want to do this randomly so we don't mess anything up.

e) Multiply our encoding by weights. (Matrix Multiply '@'). This is super efficient!!! We can feed in all 5 of our examples and simultaneously compute the activation of each neuron for each example.

f) We want 27 neurons in our first layer. So we change the shape of our weights matrix to 27x27. These neurons are do not have a bias nor a nonlinearity.

g) We are not going to have any other layers. The simplest neural net. A single layer.

h) what should these 27 outputs be? We are trying to produce a probability distribution for each character so we can predict what the next character will be. Right now, we just have some positive and negative numbers. Probabilities are positive and they sum to one. Thus, these cannot come from a neural net.

i) How do we interpret the output? The outputs are log(counts) (logits) so we just exponentiate them. Then we can use these to calculate probability. Thus, creating a distribution which we can sample to pick the next letter??

j) We then get n rows for every one of our n examples. Each row is a distribution. The neural nets assignment for how likely each one of the 27 numbers is to come next.

k) Softmax: Taking the logits, exponentiating them, and then normalizing to get a probability. Very common layer in neural nets. A way of taking the outputs of a neural net layer and creating a probability distribution from them. The outputs can be negative, positive, any number really. but the result of the softmax is a vector which is totally positive and sums to 1.

l) Each operation is easy to back propagate through and differentiable. Thus, we can train the model with gradient descent.

6. How to optimize a neural net. (**DO NOT GUESS AND CHECK. According to Andrej: "This is amateur hour"**) The big deal is that we have a loss function! We can

minimize this loss through back propagation and gradient descent!!! Booyah! Remember the algorithm for gradient descent:

a) Forward pass
b) Calculate loss
c) Zero-gradients
d) Back propagate
e) Update parameters in the opposite direction of the gradient
f) Repeat until "done" (you choose/good enough accuracy)

7. We are using the negative log likelihood because we are doing classification and not regression. Reminder: Look into this more.

8. Calculating our loss:

a) Pluck out the probabilities corresponding to our targets.
b) Take the log of each
c) Take the average
d) Negate the result
e) This is our loss function for classification

9. Zero out the gradients. You can just set them to None. (This is more efficient)

10. Now we back propagate. Reminder: We need to set requires_grad=True when we intialize our weights. We cannot back propagate if we do not do this. Now, just call, loss.backward(). AUTOMAGIC!

11. Now, W.grad tells us how each weight to each neuron influences the total loss. Thus, we now have a direction for updating the parameters. Reminder: We want to update in the opposite direction of the gradient. this looks like W += -η * W.grad where η is the learning rate.

12. When we achieve low loss, this means the network is assigning high probability to the characters that should come next, low probability to the characters that should not come next, and everything in between

13. Tangent: Optimization in this case just means getting your neural network to perform better/be more accurate.

14. **IMPORTANT:** We actually end up approaching the same loss value that we got when we just counted everything. We are just approaching the same problem from a different perspective. This is because we are not taking in anymore data than we had prior. We are still analyzing bigrams: looking at only the previous character to try and predict the next character. Note, this is only because the

bigram statistics are very straightforward and the problem is not complicated. Gradient descent learning is much more flexible and can accommodate much more complex problems. (Still make sure you analyze the complexity of what you are trying to solve and choose the approach that suits the problem best).

15. How are neural nets more flexible? We can use the same basis and fundamentals to set up our network. Rather than just feeding in one character to one layer, we can feed in multiple and have multiple layers.

16. What are the fundamentals of a neural net?

    a) The output will always be logits

    b) We take these logits through a softmax

    c) We take the output of the soft max to calculate our loss (negative log)

    d) We then backpropogate through our loss

17. The only thing that will really change is how we do our forward pass.

18. The one-hot vectors multiplied by the weights: What is actually happening is the one hot just plucks out the corresponding weight.

19. Trying to incentivize W to be closer to 0 is equivalent to smoothing. The more that we send each W to 0, the more uniform our distribution gets? This introduces what is known as regularization. We can add regularization term to our loss calculation to try and incentivize W to go to 0, while at the same time trying to get the probabilities to match what we want the model to predict. Note: Look more into regularization later! We have to choose a regularization strength. You can think of regularization as a gravitational force and the weights desire to reach a probability that we want as something growing upward. The regularization sort of resists the changing of the weights. If it is too high, then we accumulate way too much loss and the weights cannot reach a value that results in an accurate model. With the right regularization, the distribution gets smoothed slightly, but the weights are still able to achieve their desired values, resulting in a model that is accurate, not-buggy, and realistic?

Jun. 14, 2023 — Wednesday

Harvey's Notebook #4 (A lot of these notes will be direct paraphrases of Harve's notebook. You can find the notebook here: Dam n.d.

1. Cross-Correlation: A convolutional layer followed by an addition. The cross correlation of vectors w and x is $w \star x = z \in \mathbb{R}^{|x|-|w|+1}$. For example, suppose $w = [1, 2, 3]$ and $x =]1, 2, 3, 4, 5]$. Then $w \star x = [1*1+2*2+3*3, 1*2+2*3+3*4, 1*3+2*4+3*5]$

2. Convolution: In ML this most often means cross-correlation. Generally a convolution is: an integral that expresses the amount of overlap of one function g as it is shifted over another function f. In other words, we are shifting the function g over f to extract some crucial information. In cross-correlation, we are shifting w over x to see how w changes x. Kernel choice in convolution matters greatly and it all depends on the information we want to extract from x

3. How are convolutional layers beneficial with respect to simple linear transformations? Convolutional layers allow us to pick out certain features from the image. If we know an image that we are trying to classify has very specific features, we can try to define a convolutional layer that will pull out those specific features. This allows for easier classification and labeling. We can just look at the data after applying a convolutional layer and say "Hey that looks like (or does not) this...".

4. As a side note, convolutions can help us detect edges in images, filter images, sharpen images, and much more

5. Convolutional network layers have many kernels of different shapes and strides (how many pixels to move the kernel over after each i or j step. The kernels are learnable, so their elements serve as parameters in the neural network

6. Paraphrase: "Each kernel learns to detect a certain feature in the input, all combining to distill the information that we want from the input. Earlier layers will detect more simple shapes, while later layers will detect more sophisticate things such as shapes or faces."

7. PyTorch Modules are functions that keep track of their parameters and gradients.

8. nn.functional.conv2d applies a 2D Convolution over an input image composed of several input planes

9. Hyperparameters are parameters that we set before training which we do not update as we train. (in some cases you actually might. For example, you may scale the learning rate as you train so you can focus on the finer details later on in the training.)

10. We then need to make a dataloader out of the original images and our target images. (Yes, we computed the target images when we called nn.functional.conv2d).

11. We will be using stochastic gradient descent (SGD) with torch.optim.SGD

12. Upon loading each batch:

    a) Zero all gradients (Note: **You must zero out the gradients before you backpropagate!!**)

    b) Forward pass

    c) Calculate loss

d) Backpropagate

e) Update Parameters and the Learning Rate

13. One of the more common and appropriate loss functions for image classification is Cross-Entropy Loss. Reminder: Look this up later!

Harvey's Lecture

1. Two-Layer Perceptron: Linear, Relu, Linear

Mu's Lecture: Malware and Machine Learning

1. Question: How can we classify malware?
2. The total amount of malware is increasing. We discover new malware every year.
3. Example: rootkit -> Can tamper with the OS Kernel and hide it self from the user. Gives the attacker root access to the user's system.
4. Viruses take up 9% of total malware
5. What is a virus: Replicates itself by modifying other programs and inserting its own code. Must find and live within a host. By nature, it is parasitic.
6. Propagation: Opportunistic through user interaction: running an app, booting the system
7. When it runs it looks for opportunity for infection. Looks for its own chance to run.
8. You can embed code in pdf's, Microsoft Office documents (macros)
9. Pseudocode for virus: **Look at Mu's slides and insert picture later**
10. Viruses only infect useful targets and only ones that have not already been infected. Then needs to modify the code to make sure that the malicious code will actually run
11. One way that we can do this is to prepend the virus to the program code. This is too obvious! The code is easy to identify
12. Alternate way: Append virus code to end of the original program. Also add code at the beginning to make sure the virus code runs first
13. First virus: Creeper in 1971. Not a malicious virus. Just meant to gague the size of the ARAPNET
14. How do we detect malware?

15. Read Manually? Can be difficult because we do not have source code. We then need to reverse engineer it. (No symbol information). Not all languages need to compile, they just need an interpreter. Thus, we can still read some of the source code: js, Python. Still hard because of obfuscation technique
16. Obfuscation is the deliberate act of creating source or machine code that is difficult for humans to read and understand
17. Signature Detection: Ability to detect some malicious signatures
18. If we know that for a certain type of malware there is a specific signature, we can recognize it. These signature detection services are currently provided in all virus detection services
19. Polymorphism: A technique to evade signature based detection. Polymorphic code is the code that uses a polymorphic engine to mutate while keeping the original algorithm intact. Encryption is the most common method to hide code
20. Now, we just encrypt the code and insert the encrypted copy into the victim file. A weak, but simple and fast encryption technique works just fine
21. When the injected code runs, it needs to decrypt and then run. How?
22. insert decrypter and key in plain text and then the encrypted code
23. The injected code also contains en encryptor so it can encrypt itself when it infects a new host. Obviously, you use a different key
24. Defender can just use the decryptor and key to decrypt the malicious code and calssify it
25. Metamorphism: Code that when run outputs a logically equivalent version of its own code under some interpretation
26. Everytime the virus propagates, it generates syntatically different code.
27. How?
    a) Add NOPs
    b) Add useless instructions and loops within code segments
    c) Change register to use
    d) Change order of conditional code
    e) Reorder operations not dependent on one another
    f) Replace one low-level algo with another
    g) Remove some do-nothing padding and replace with different do-nothing padding

28. With metamorphism, the original signature is completely gone.
29. Semantics aware malware detection: The fundamental deficiency in the pattern-matching approach is that it ignores the semantics of the code
30. Semantic aware malware detection actually cares about a what a program does, not just the outcome. What are we actually looking?
31. We build a model on top of the code to remove the unreliable parts: register values, # of nops, things that can change with each program execution
32. we parse the code to see what it is actually trying to do and in what order. we can use this high level model and the patterns to perform detection
33. how do we build this model? Static and dynamic analysis
    a) Static Analysis: Just looking at the code to observe what it actually doing and what it accomplishes in the end. Very scalable. Maybe not super sophisticated. You can only observe statically generate code. You cannot see the code that can only be generated when the program runs
    b) Self Modifying code: Code that can change itself. You usually need to change the permissions of address memory so you can modify the code dynamically. So, static analysis is not great for detecting malware especially in binary programs
    c) Dynamic Analysis: Analyzing the code as it runs
    d) Can do this with a debugger -> Simple. Cannot do this with code that does not have symbols
    e) You can also use emulators: VMWare, VirtualBox, other containers. You can just run the malware in the dedicated environment and then see what happened. Super useful so you do not mess with you own computer -> called Sandboxing. We can also have a complete view of the program. You can observe the behavior because you are outside of the emulator.
    f) Dynamic analyzers will run the program and use the outcomes to generate a signature and then use that signature to classify malware. But, this is the defacto approach so malware users want to avoid this.
    g) How?
        i. Malware users can detect the presence of a debugger. Then they will hide the malicious code
        ii. Emulators can also be easily detected through environment variables. Can also check the difference in behavior of program through the emulator and the program on the actual hardware. Timing can also be a factor. Typically much faster on the CPU than in the VM
    h) What is the solution?

   i. Run malware on bare-metal. All CPU's have virtualization techniques. CPUs can record every single line of code that was run. That way we can recover the behavior of the malware

i) Now, back to the project. We want to understand the more reliable bits of malware to train a classifier
   i. Polymorphism: Consider code pattern not bits pattern
   ii. Metamorphism: Consider what program does and not just the outcome
   iii. What features or code captures can we detect?

j) Let's build a malware classifer - a multi-class classifier. The features are already collected for you - On Canvas! :)
   i. "mov" feature: how many "mov" instructions one malware may contain
   ii. ".text" sections: how many ".text" sections one malware may contain

k) Task: Train a malware classifier to automatically recognize which class a malware instance belongs to

Xinyi's talk on FP

a) IEEE has defined 5 exceptions with FP numbers
   i. Invalid
   ii. Div by 0
   iii. Overflow
   iv. Underflow
   v. Enexact

b) NVIDIA GPU's do not trap exceptions! They should be caught!

c) Exception reports on GPUs
   i. NaNs in Neural Network
   ii. NaNs in Linear Algebra Solver

d) We need a tool that is fast, at binary level, and helps diagnose issues on the GPU

Jun. 15, 2023 — Thursday

Sadayappan's Talk

1. Parallelism and optimization

Jun. 20, 2023 — Tuesday

Jeff Phillips Talk:

1. Morning activity about buying and selling data

2. Thinking about predicting data from a set of inputs as a function
3. Mean squared error can overfit the data: exaggerate any outliers. Can be likely to generalize on new data that we haven't seen yet. When we see new data, the loss function may change drastically to fit this new data
4. How do we overcome this? Have training and testing data be two different sets of data. Quanitfying outliers and removing them from the data set.
5. How do you find and determine what is and what is not an outlier? There are a bunch of heuristics about how to do this. There have been new algos that have come out about training models and finding the mean of higher dimensional data to deal with outliers. These algos can actually work well with up to 30% outliers in the data.
6. How do we gather testing data? We randomly sample the data and do an 80/20 split between testing and training data.
7. Cross-Validation: Using a testing and training split to analyze how well the model is performing
8. The data needs to bee IID in order to make this work: Independent, Identitical data.
9. Do not put everything into training, otherwise you can overfit and not actually perform well
10. Where can this get better? What is wrong about this?

Jun. 21, 2023 — Wednesday

Started by reading d2l.ai. See d2l.ai Overleaf for notes.
TREU 2023:
Hannah Cohoon's talk

1. Human Centered Computing:
2. Do artifacts have politics? Technology or artifacts do take a stance. Legislation can always be changed, but it can be hard to tear down a bridge once it is up.
3. The bridges in Long Island, New York are a classic example of biased technology
4. HCI gives us a perspective and a framework for decisions that help us avoid negative technology and technology that gatekeeps.
5. Value conflict: When a designer's values are in conflict with the users values. The user may also have values that can conflict with their own values
6.

References

Dam, Harvey (n.d.). *Reu 2023*. URL: https://github.com/damtharvey/reu2023.

Karpathy, Andrej (n.d.[a]). *The spelled-out intro to language modeling: building make-more*. URL: https://www.youtube.com/watch?v=PaCmpygFfXo&list=PLAqhIrjkxbuWI23v9cThsA9GvCAUhRvKZ&index=2&t=2s&ab_channel=AndrejKarpathy.

——— (n.d.[b]). *The spelled-out intro to neural networks and backpropagation: building micrograd*. URL: https://www.youtube.com/watch?v=VMj-3S1tku0&t=5246s&ab_channel=AndrejKarpathy.

Smola, Aston Zhang Zachary C. Lipton Mu Li Alexander J. (n.d.). *Dive into Deep Learning*. URL: https://d2l.ai.