# Basic ML

Summer 2023

September 2, 2023

**Name**: Richard Xu
**Code**: https://github.com/rxu183/python-playground/tree/main/GAN

## 1 Data Acquistions

So, before diving into the math and the models, you first need to find a way to get some interesting data for your project and preprocess it. As the saying goes, "Garbage in, garbage out." So be sure to get your data from good sources and clean it thoroughly. Here, I'll discuss a few data collection methods/strategies that I used when creating this guide (you can skip this section and come back when you need it, or read it straight through). Here are some sources:

1. Kaggle. This feels like cheating, but people often put a lot of high quality datasets on this data science hub. When you're first starting out and don't want to worry about the hassle of data collection, this is an excellent source to focus on learning the models. They also have nifty little competitions and prizes if you create an account.

2. Web Scraping! The internet is amazing, and there's a lot of cool data out there. This is what I cover and implement down below.

3. Universities/Research - oftentimes, papers make the data they trained on available alongside their paper so that others can verify their results, and this is particularly useful when you're trying to recreate their models to get a better understanding of what they're talking about. These datasets might also be hosted by universities. UC Irvine offers some pretty good ones for machine learning (it's where I got my Car Prices one you see below).

4. You. I cannot offer advice on this, but if you have the wherewithal and passion to go out and collect data points, you will certainly get original data, and perhaps a more interesting project. Use your imagination to find data on something you've always found interesting!

So, how do you build a web scraper anyhow? What is it? A web scraper is just a tool we use to extract data from the internet (as its name would suggest).

Basically, we rely on two primary Python libraries to do our form of pretty easy web scraping - BeautifulSoup4, and requests, and I also used wget (available on mac via brew install wget, or linux via apt/apt-get wget, or maybe even Windows via WSL2 and wget (though I haven't tried it)). wget is a very fast downloader, but we use requests to get the html, then Beautiful Soup4 to search for the relevant html tags (usually in the form of an <a class = "some_class" href = "link_of_interest"> or something). All this requires is opening developer tools, and then poking around. However, websites can have quite a few nested tags, so a better method is to search for the content you're trying to scrape, and then right-click, and then inspect to jump to the relevant section of the website.

This obviously varies a lot depending on what information you're trying to scrape, and from website, so it'll probably take a little of experimentation to get it to work. Another word of warning - if the site you're trying to scrape from has Cloudflare or another security/captcha software/they don't want bots

on their site, you can really dive down the rabbit hole of trying to use authentication to get by the website, but usually it's not worth it, and can cause legal troubles depending on terms of use/the like. If you really wanted to scrape data from one of those websites, I would recommend using some form of autoclicker to record your actions - for example you could sit through one page of right clicking and then saving as, and then let it continuously do that (and automatically hitting the next button). Read carefully before scraping!

Obviously, programmatically doing it is faster and more reliable (page numbers might reach double digits, or even three digits, and then the location where you clicked the "next" button will be off, etc.), and it's a fun little thing to build for yourself.

I built my web scraper to run on Ebay. It's pretty simple - i just searched for the link of shoes, refreshed so that I got the link referring directly to the search indices as opposed to the weird hash that shows up when you search directly.

# 2 Linear Regression

Logistic Function =

$$F : A \to B, A, \in \mathbb{R}, B \in \mathbb{R} | b \in B \implies 0 \leq b \leq 1$$

More explicitly:

$$sigmoid = \frac{1}{1 + exp(-h_\theta^-(x))}$$

Gradient Descent Algorithm:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Where $\alpha$ is the learning rate, and $\theta_j$ is the jth parameter being changed

For example, consider the example where the cost function we're dealing with, $J(\theta)$ turns out to be the mean squared error.

Note also that $x_j^i$ does not refer to exponentiating to the index $i$, but rather that it refers to the $i$th training example used. In short, we are using $i$ training examples, each example we suppose is being determined by $j$ parameters.

$$J(\theta) = \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Let's first consider how we'd find the term $\frac{\partial}{\partial \theta_j} J(\theta)$:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j} \sum_{i=1}^{n} [\sum_{j=0}^{m} (x_j^i \theta_j^i) - y^i]^2$$

Let

$$u = \sum_{j=0}^{m} (x_j^i \theta_j^i) - y^i$$

Then, note also that the derivative of a summation is simply equal to the summation of those derivatives - in other words,

$$\frac{\partial}{\partial \theta_j} \sum_{i=1}^{n} u^2 = \sum_{i=1}^{n} (\frac{\partial}{\partial \theta_j} u^2) = \sum_{i=1}^{n} (2u \frac{\partial}{\partial \theta_j} u)$$

Now, let's consider:

$$\frac{\partial}{\partial \theta_j} u$$

Substitute $u$ back in:

$$\frac{\partial}{\partial \theta_j} u = \frac{\partial}{\partial \theta_j} \sum_{j=0}^{m} (x_j^i \theta_j^i) - y^i$$

Note that all terms in the summation are constant relative to $\theta_j$ (including $-y^i$ - write out the summation if this seems confusing (I certainly did on paper before typing this up)), and therefore,

$$\frac{\partial}{\partial \theta_j} u = x_j^i$$

Filling everything back out, we get:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \sum_{i=1}^{n} [2x_j^i (\sum_{j=0}^{m} x_j^i \theta_j^i) - y^i]$$

Returning to our original gradient descent formula, we can fill it out as follows:

$$\theta_j = \theta_j - \alpha \sum_{i=1}^{n} [2x_j^i (\sum_{j=0}^{m} x_j^i \theta_j^i) - y^i]$$

Now, let's see another way to work with linear regression. Consider the vectorized (and cleaner) way of arriving at this solution in one step. While we won't dive in via pseudo-inverses and all that, that certainly is a possibility to solve this problem more thoroughly.

First of all, note that the linear least squares problem can be formulated as finding the value of the parameters $\theta$ such that the expression:

$$||y - A\theta||_2^2$$

is minimized, where $y \in \mathbb{R}^n$ represents the vector of $y\_actual$ data values (assume we have $n$ total data points we're trying to fit), and $\theta$ represents the column vector $(m, b)^T$, where $m$ denotes the slope of our target line, and $b$ denotes the intercept of that line (just traditional $y = mx + b$).

Matrix $A \in \mathbb{R}^{nx2}$ on the other hand represents a matrix with the first column being the $x_{coordinates}$, and the second column just being a column of 1's. Note that when you expand these columns and multiply them by each other, you get that we're trying to minimize the square of the two norm of $y_{iactual} - (mx_i + b)$, which conceptually makes a lot of sense - it's the as our error function above! (there are mathematical reasons why we select the two norm specifically, but honestly, I forgot them - I think it's partly because of this next nice property we'll apply to go from here:).

Namely, that:

$$||y - A\theta||_2^2 = (y - A\theta)^T (y - A\theta)$$

. From here, we can apply traditional logic from single variable calculus and look for extrema. Note that we don't know whether any extrema returned will be a local minimum, maximum, or saddle point - but we can figure that out via the Hessian/Second Derivative Test, which we will apply shortly thereafter.

First though, let's expand our expression by expanding our transpose, and multiplifying through:

$$\begin{aligned} J(\theta) &= (y - A\theta)^T (y - A\theta) \\ &= (y^T - ((A\theta)^T)(y - A\theta) \\ &= (y^T - \theta^T A^T)(y - A\theta) \\ &= y^T y - \theta^T A^T y - y^T A\theta + \theta^T A^T A\theta \end{aligned}$$

Then, we take the gradient of the above. Note that taking a gradient means, separately, a bunch of partial derivatives with respect to each individual varaible. Note also that since we're taking gradients with respect to $m$, and $b$, in our $\theta$ matrix, terms without any of the input variables will disappear (namely, $y^T y$).

Let's approach the rest of the expression term by term -

$$\theta^T A^T y = [m, b] \begin{bmatrix} x_1 & ... & x_n \\ 1 & ... & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ ... \\ y_n \end{bmatrix} = m \sum_{i=1}^{n} x_i y_i + b \sum_{i=1}^{n} y_i$$

Then, recall that if some function $x$ takes in a vector of inputs, the following holds:

$$\nabla f(\vec{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ ... \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

More relevantly,

$$\nabla f(\theta) = \begin{bmatrix} \frac{\partial f}{\partial m} \\ \frac{\partial f}{\partial b} \end{bmatrix}$$

Now it becomes clear that given the other summation will poof when taking this partial respective to one of the variables, and that we'll get:

$$\nabla \theta^T A^T y = \begin{bmatrix} \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i \end{bmatrix}$$

Then, notice that we can rewrite this as:

$$= A^T y$$

Our next expression seems pretty familiar...

$$\nabla y^T A\theta = \begin{bmatrix} y_1 & ... & y_n \end{bmatrix} \begin{bmatrix} x_1 & 1 \\ ... & ... \\ x_n & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = m \sum_{i=1}^{n} x_i y_i + b \sum_{i=1}^{n} y_i$$

At this point, we can probably realize we can safely take the derivative based on the vector $\theta$, and we also interestingly discovered that there are some transpose/order shuffling that's allowed because the result of this matrix multiplication is a scalar. Anyhow, though, we get what we got above:

$$\nabla y^T A\theta = A^T y$$

Ok, on to the last one (Deep Breath!)"

$$\theta^T A^T A\theta = \begin{bmatrix} m & b \end{bmatrix} \begin{bmatrix} x_1 & ... & x_n \\ 1 & ... & 1 \end{bmatrix} \begin{bmatrix} x_1 & 1 \\ ... & ... \\ x_n & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$= \begin{bmatrix} mx_1 + b & ... & mx_n + b \end{bmatrix} \begin{bmatrix} x_1 & 1 \\ ... & ... \\ x_n & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{i=1}^{n} (mx_i + b)x_i & \sum_{i=1}^{n} (mx_i + b) \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$= m \sum_{i=1}^{n} (mx_i + b)x_i + b \sum_{i=1}^{n} (mx_i + b)$$

Phew! Ok, now let's take the gradient of that (which unfortunately will be much more nasty than before given that the sums aren't so nicely independent). Let's first start by defining the above function as $g(m, b)$. For this one, it'll also be easier to put the $m$ and $b$ inside the summations, so let's go ahead and do that.

$$g(m, b) = \sum_{i=1}^{n} (m^2 x_i + mb)x_i + \sum_{i=1}^{n} (bmx_i + b^2)$$

$$= \sum_{i=1}^{n} (m^2 x_i^2 + bmx_i + bmx_i + b^2)$$

Then, we take the partial with respect to $m$!

$$\frac{\partial g}{\partial m} = \frac{\partial}{\partial m} \sum_{i=1}^{n} (m^2 x_i^2 + bmx_i + bmx_i + b^2)$$

$$= \sum_{i=1}^{n}(2mx_i^2 + 2bx_i)$$

Then, we take the partial with respect to $b$!

$$\frac{\partial g}{\partial b} = \frac{\partial}{\partial b}\sum_{i=1}^{n}(m^2x_i^2 + bmx_i + bmx_i + b^2)$$

$$= \sum_{i=1}^{n}(2mx_i + 2b)$$

Finally, putting it all together:

$$\begin{bmatrix} \sum_{i=1}^{n}(2mx_i^2 + 2bx_i) \\ \sum_{i=1}^{n}(2mx_i + 2b) \end{bmatrix}$$

Now, uh, this is quite difficult, but let's analyze how these puzzle pieces fit together. So, it looks pretty apparent that we can take out a factor of $\theta$ without too much difficulty, so let's go and do that, and while we're at it, we might as well take that scalar multiple 2 out front.

$$2\begin{bmatrix} \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} bx_i \\ \sum_{i=1}^{n} mx_i & \sum_{i=1}^{n} 1 \end{bmatrix}\begin{bmatrix} m \\ b \end{bmatrix}$$

Then, let's pick apart these summations - to get a 2x2 with all of these n-length sums, it seems likely that we'll be doing something along the lines of a $2 \times n$ by $n \times 2$ multiplication. Note also the suspicious square terms. Indeed, it seems we can decompose this into:

$$2\begin{bmatrix} x_1 & ... & x_n \\ 1 & ... & 1 \end{bmatrix}\begin{bmatrix} x_1 & 1 \\ ... & ... \\ x_n & 1 \end{bmatrix}\begin{bmatrix} m \\ b \end{bmatrix}$$

$$= 2A^TA\theta$$

Since there is no factor of $\theta$, this one just zeros out.

$$\nabla y^Ty = 0$$

Finally, we can write our original gradient as:

$$\nabla\left(y^Ty - \theta^TA^Ty - y^TA\theta + \theta^TA^TA\theta\right) = 2A^TA\theta - 2A^Ty$$

Phew! Now, we set this gradient equal to 0, and find that our parameters are equal to:

$$0 = A^TA\theta - A^Ty$$

$$A^Ty = A^TA\theta$$

$$\theta = (A^TA)^{-1}A^Ty$$

Ok, now that we've finally finished laying the groundwork for all of this, it's a relatively simple matter to apply this formula to actual data. Head on over to https://calcofi.org/data/oceanographic-data/, download the "bottle" database, and then read it in with pandas, and then apply this formula based on two columns we're interested in - temperature and salinity to get the following:

Coefficients:

$$\theta = \begin{bmatrix} -0.05571849096000081 \\ 34.44411969121523 \end{bmatrix},$$

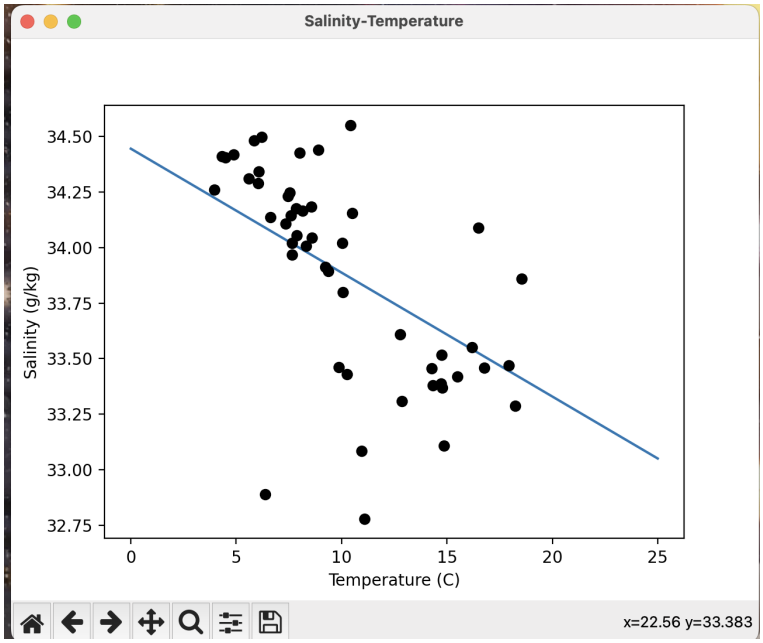


Figure 1: Salinity-Temperature line, plotted together with 50 random data points.

## 3 Neural Networks

So, we've skipped several steps, but let's understand for ourselves what neural networks are, from the ground up.

These involve non-linear functions. I.e. we are leaving the realm of $y = mx + b$. However, more than just that, we are leaving the realm of $y = ax^2 + bx + c$, and actually entering the realm of more like $y = a^2x^2 + b^24x + cx^(5/2) + a^3x^3$ – both our parameters, (our a's), and our (x's) are non-linear. The reason why neural networks are so powerful are because of something called the Universal Approximation Theorem. The idea is, that if I have enough of these neurons and their weights are set well, I can represent a lot of relevant/useful functions.

There is a reason why people refer to these models as black boxes – it's hard to imagine how these formulas are created, and indeed, we don't *really* know what these "discovered features"/weights mean and/or how they connect to the real world - we just use the power of the computer and some general principles, and then tell the computer to find the best solution.

In general, here are the four steps we need to run deep learning on our model:

1. Define a parametrization (model)
2. Define a loss function (the function that scores how well your function does relative to expected output)
3. Write the backpropagation algorithm that calculates your gradients. (math to give you information on how to update)
4. Run mini-batch stochastic gradient descent or some other optimizer to update the parameters (weights) of your model.

So, let's explain a neural network before moving on to the fancier ones.
First of all, a neuron is primarily a matrix multiplication with a few steps in sequence:

1. An input vector, known as a vector of features is passed into the neuron.

2. The input is multiplied by a weight matrix.
3. Add a constant to your result (called a bias; if you want to make it harder or easier to activate). Note that this is a learned parameter, and therefore your model can optimize it depending on the inputs/outputs, like the multiplicative parameters.
4. The result of the matrix multiplication is passed through an activation function.
5. You have your result vector! Pass it into the next intermediate layer, or pass it into your result layer, or even just return it directly (most models have more than one layer though).

That could be confusing. So let's consider the example of wanting to predict car price from an array of features.

You can see the full datset here: https://archive.ics.uci.edu/dataset/10/automobile

Let's say I believe that a car's price was dependent on a few features - the fuel a car uses, the number of doors it has, the type of car it is (e.g. convertible vs sedan), the type of drive it has, among others.

So, I have a total of 26 input features (note this would normally be a singular row vector, as in figure below but for all applications, you usually have (a lot of) data points where the values of the features and their corresponding output are known – ie there are many examples of all 26 (or some, and then you have to fill in data with mean/median/other method) of these features for each car sold ).

| | price | fueltype | aspiration | doornumber | carbody | drivewheel | enginelocation | cylindernumber | enginetype | fuelsystem | symboling | ... | carwidth | carheight | curbweight | enginesize | boreratio | stroke | compressionratio | horsepower | peakrpm | citympg | highwaympg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 13495.0 | gas | std | two | convertible | rwd | front | four | dohc | mpfi | 3 | ... | 64.1 | 48.8 | 2548 | 130 | 3.47 | 2.68 | 9.0 | 111 | 5000 | 21 | 27 |
| 1 | 16500.0 | gas | std | two | convertible | rwd | front | four | dohc | mpfi | 3 | ... | 64.1 | 48.8 | 2548 | 130 | 3.47 | 2.68 | 9.0 | 111 | 5000 | 21 | 27 |
| 2 | 16500.0 | gas | std | two | hatchback | rwd | front | six | ohcv | mpfi | 1 | ... | 65.5 | 52.4 | 2823 | 152 | 2.68 | 3.47 | 9.0 | 154 | 5000 | 19 | 26 |
| 3 | 13950.0 | gas | std | four | sedan | fwd | front | four | ohc | mpfi | 2 | ... | 66.2 | 54.3 | 2337 | 109 | 3.19 | 3.40 | 10.0 | 102 | 5500 | 24 | 30 |
| 4 | 17450.0 | gas | std | four | sedan | 4wd | front | five | ohc | mpfi | 2 | ... | 66.4 | 54.3 | 2824 | 136 | 3.19 | 3.40 | 8.0 | 115 | 5500 | 18 | 22 |
| .. | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 200 | 16845.0 | gas | std | four | sedan | rwd | front | four | ohc | mpfi | -1 | ... | 68.9 | 55.5 | 2952 | 141 | 3.78 | 3.15 | 9.5 | 114 | 5400 | 23 | 28 |
| 201 | 19045.0 | gas | turbo | four | sedan | rwd | front | four | ohc | mpfi | -1 | ... | 68.8 | 55.5 | 3049 | 141 | 3.78 | 3.15 | 8.7 | 160 | 5300 | 19 | 25 |
| 202 | 21485.0 | gas | std | four | sedan | rwd | front | six | ohcv | mpfi | -1 | ... | 68.9 | 55.5 | 3012 | 173 | 3.58 | 2.87 | 8.8 | 134 | 5500 | 18 | 23 |
| 203 | 22470.0 | diesel | turbo | four | sedan | rwd | front | six | ohc | idi | -1 | ... | 68.9 | 55.5 | 3217 | 145 | 3.01 | 3.40 | 23.0 | 106 | 4800 | 26 | 27 |
| 204 | 22625.0 | gas | turbo | four | sedan | rwd | front | four | ohc | mpfi | -1 | ... | 68.9 | 55.5 | 3062 | 141 | 3.78 | 3.15 | 9.5 | 114 | 5400 | 19 | 25 |

Figure 2: One row represents one training example. Interestingly, this extends to images as well, which can be harder to visualize. Also shows pandas output from reading in the csv.

In other words, each feature gets one column in my input matrix (so a specific example sale of a car would be a single row in this matrix).

| Fuel Type | Door Number | Type of Drive | Car Body Type ... |
|---|---|---|---|
| Gasoline | 2 | Rear Wheel Drive | Convertible |
| Gasoline | 2 | Rear Wheel Drive | Convertible |
| Diesel | 4 | Rear Wheel Drive | Sedan |
| Gasoline | 4 | Four Wheel Drive | Sedan |

Now, a problem immediately becomes apparent - how do I deal with this data that is in the form of strings when all of neural nets require matrices? Well, in our data set, we can map our inputs to boolean vectors - e.g. Gasoline as type [1, 0] , and Diesel as type [0, 1], RWD/FWD similarly, and extending this to as many types of data as I have, e.g. if I have four types of data, I'd use [0, 0, 1, 0] to encode the presence of the third element. This is known as one-hot encoding.

| | price | symboling | wheelbase | carlength | carwidth | carheight | curbweight | enginesize | boreratio | ... | enginetype_rotor | fuelsystem_1bbl | fuelsystem_2bbl | fuelsystem_4bbl | fuelsystem_idi | fuelsystem_mfi | fuelsystem_mpfi | fuelsystem_spdi | fuelsystem_spfi |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 13495.0 | 3 | 88.6 | 168.8 | 64.1 | 48.8 | 2548 | 130 | 3.47 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 16500.0 | 3 | 88.6 | 168.8 | 64.1 | 48.8 | 2548 | 130 | 3.47 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | 16500.0 | 1 | 94.5 | 171.2 | 65.5 | 52.4 | 2823 | 152 | 2.68 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | 13950.0 | 2 | 99.8 | 176.6 | 66.2 | 54.3 | 2337 | 109 | 3.19 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 4 | 17450.0 | 2 | 99.4 | 176.6 | 66.4 | 54.3 | 2824 | 136 | 3.19 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| .. | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 200 | 16845.0 | -1 | 109.1 | 188.8 | 68.9 | 55.5 | 2952 | 141 | 3.78 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 201 | 19045.0 | -1 | 109.1 | 188.8 | 68.8 | 55.5 | 3049 | 141 | 3.78 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 202 | 21485.0 | -1 | 109.1 | 188.8 | 68.9 | 55.5 | 3012 | 173 | 3.58 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 203 | 22470.0 | -1 | 109.1 | 188.8 | 68.9 | 55.5 | 3217 | 145 | 3.01 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 204 | 22625.0 | -1 | 109.1 | 188.8 | 68.9 | 55.5 | 3062 | 141 | 3.78 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Figure 3: One-hot encoded data

There exist other techniques for natural langauge processing where you are trying to understand the meaning behind a sentence, e.g. bag of words, or word embeddings, but this one will do for now.

Right, so, now, we just dive into the data. Here, I've removed the text and index columns, and shifted the price column over to index 0 so that it's readily available and we can sift it out without much difficulty.

Then, we use pandas to load in our data, use get_dummies to turn the categorical data into the vectors I mentioned earlier (they form new columns - our individual entries are not vectors), and turn our target column into its own tensor, and turn the rest into another tensor from pytorch.

Pause. What is a tensor anyhow? A tensor is simply put, a multi-dimensional array. An array of arrays of arrays ..., but optimized for GPU/TPU parallelization/just more efficient math, which neural network training benefits from.

So, now, we need to pick which machine library you want to learn - there's tensorflow, and there's pytorch. Tensorflow is developed by Google, used more in industry and is a little more Linux/server oriented than Windows OS oriented in particular (you have to use WSL2 for GPU, and that's a pain to configure) and PyTorch developed by Meta and used more in academia, so there's that. We'll use PyTorch for now.

First, note that in that section at the top we included some boilerplate to use GPU or Metal MacOS backend, or CPU if all else fails. Then, we begin creating our model.

We subclass the pytorch.nn.module to create our neural network, and we call the super's init, and now finally we can start adding layers to our basic neural network!

So there are several types of layers - linear/dense/fully-connected are the ones we're working with, and is what you'll hear it be called.

In our case, I'll just use a 64 node first layer, and a 32 node second layer, given that we have a total of 52 input features (note that this is up from the original 26 after the conversion into categorical variables), and you roughly want at least as many neurons as inputs (because theoretically all inputs could be very independent and not be "groupable" into higher order features).

So, short aside, you see on the news and everywhere boasting that ChatGPT has 300 billion parameters. How many does our model have? Let's explain how to calculate this number (for dense layers, at least).

1. The easy way - with Keras API in tensorflow: model_name.summary() - This will directly give you the number of parameters your model has, sorted into trainable vs nonparameters, broken down by layers. It's very useful.

2. The "easy way" with pytorch. Sadly, there is no such one-stop function as in tensorflow. But, you can view how many parameters each model has (numel does not distinguish between trainable/untrainable). Supposedly, the following code snippet does the trick: total_params = sum(p.numel() for p in model_name.parameters())

3. Manually! For our example, we have 52 inputs feeding into 64 neurons. We expect 32 outputs from this first layer to hand off to the second. Every input goes into every neuron as a column vector (note, however, that in practice/numpy, you will see it be input as a row vector - it's more of a academia standard to use column vectors in situations like these rather than row vectors, and most all papers will use column vectors) $x \in \mathbb{R}^{52}$, and then we will have a big matrix $W$, containing the weights for this neuron. How many weights? Well, based on matrix multiplication rules, if I want 32 outputs, e.g. an output $z \in \mathbb{R}^{32}$, I need a matrix $W \in \mathbb{R}^{32*52}$, and then, I multiply $Wx$ to get mid-way vector $t \in \mathbb{R}^{32}$, but we're not done yet! We need our $+B$ (bias term), which is present only at every neuron, so the output vector of our layer $z = t + B$. So, on to the multiplication - in the first layer, I have 52 * 32 = 1664 $W_{ij}$ parameters, and 32 bias parameters, for a total of 1696 parameters in this layer! My second layer on the other hand, only has 32 inputs * 1 output + 32 bias terms = 64 parameters. My model in total will therefore have 1758 trainable parameters.

Now, after each layer of neurons we pass through, we want to pass the input through our non-linear activation function. Why do we need to do this every layer as opposed to just once at the end? Well, consider the following two layer model (note all variables below are matrices/vectors):

$$W_1 x + b_1 = z$$

$$W_2 z + b_2 = y$$

Now, plug in z into the second equation:

$$W_2(W_1x + b_1) + b_2 = y$$

$$W_2W_1x + W_2b_1 + b_2 = y$$

Note, however, $W_2b_1$ is a constant vector the same size as $b_2$, and we can effectively reparametrize our two layers into one layer as follows:

$$W_3 = W_2W_1x, b_3 = W_2b_1 + b_2$$

$$W_3x + b_3 = y$$

Thus, we need our nifty non-linear activation function after every neuron to avoid just collapsing the model down to a singular hidden layer (not that a single hidden layer is some pushover model, no, rather according to https://openreview.net/forum?id=hfUJ4ShyDEU, a single hidden layer is all the complexity needed to achieve Universal Approximation, or the Full Power of Neural Networks (in theory, anyway)).

Now, wait, let's talk about activation functions. We've kind of seen why they're important - they stop our model from just breaking down to just fitting $Wx + b = y$ due to their non-linearity, but what sorts of functions do I use? When do I use them?

Well, there are a wide variety of these functions - ReLU, Leaky ReLU, Softmax, etc. We'll go over them more thoroughly later, along with different loss functions, and different optimizers, and other types of layers you can use when creating the parametrization of your model. For now, we use the ReLU function, which looks like the following:

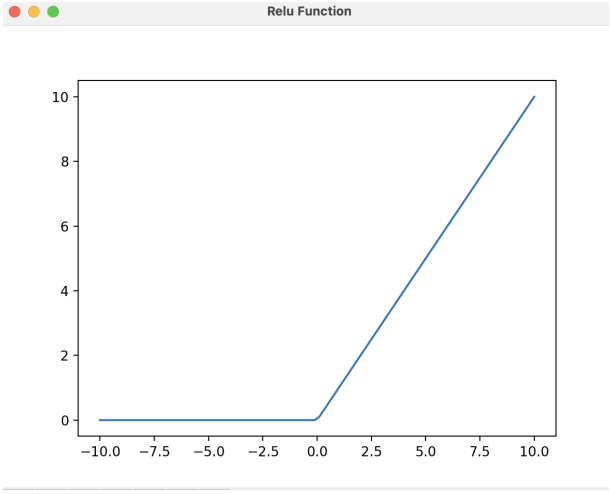


Figure 4: Shows relu function.

Yes, it's basically linear and $y = x$, but the fact that negative values flatten out at 0 is actually quite powerful, because for example, the parameters can be trained to threshold certain inputs with the help of a bias. Imagine simplified taxes, for a moment. I only consider taxing people with incomes > $5000, for example, and from then on it's a linear increase in amount taxed with income, but incomes below that aren't considered. This isn't really a direct analog, but one can see how the non-linearity might be very powerful (it's essentially an if-statement, really).

Note that our activation layer is an ordinary ReLU, rather than a Leaky ReLU in order to guarantee positive values - it wouldn't make sense for there to be a negative price of a car, after all! In general, however, your activation function of the final layer will often be different from the rest of the layers in your model, as it's no longer focused on transmitting information about features detected in this layer to future layers, but rather on converting the derived information into something useful.

Now, that we understand how these two layers are created (and can marvel at the power contained in

those few lines!), we can return to PyTorch. So, next we define our forward function to actually utilize those layers we've put in so much effort to create.

In our forward function, we receive the inputs we were interested in, and call each layer individually, updating the variable we use to carry all this information around each time, until the vector has passed through our whole model and we can return the result.

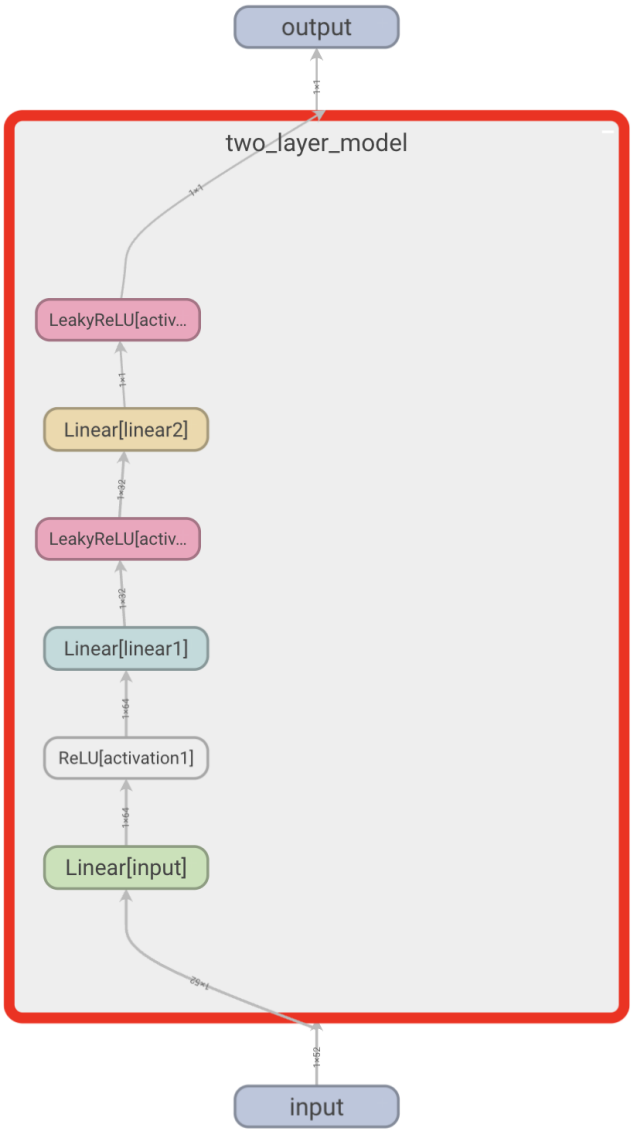


Figure 5: Model and its layers, as presented by Tensorboard

Then, we can actually instantiate an instance of our model with a simple line along the lines of myModelObject = my_model(). Afterwards, in my code, I go through and create a tensorboard object to help visualize the results, and then create our loss/cost function! There are again, many choices of loss functions depending on the task, e.g. MSE for regression, binary cross-entropy for binary classification, etc. For this example dataset, MSE will serve our purposes well enough, and PyTorch has it readily available.

Next, we need to pick an optimizer. The optimizer is entirely analagous to the little update formula given at the start of the document in our introduction to ML via linear regression! Adam - short for

adaptive moment estimation - is generally the goto optimizer of choice for most tasks, with a learning rate of roughly $1e^{-4}$.

The linear regression derivation up top is really useful in showing how neural networks and linear regression are made up of the same simple components, just with different choices. In fact, the SGD optimizer we used up top totally could've been used here, as well.

After selecting both of these key components to the model, we can write the training and testing loops to actually go through and calculate those gradients and therefore figure out how to update our parameters (which initially are completely random numbers).

Ok, now in this case we simply call our PyTorch function and have it go and do the backpropagation behind the scenes, nothing really necessary for us to do. But wait, isn't it about time we understood how backpropagation works for ourselves? I concur - so let's go through a worked example of backpropagation example:

*INSERTBACKPROPEXAMPLE*

Phew! Now, finally, we are able to appreciate the complexity hidden behind that single line of PyTorch!

The training loop is certainly worth spending some time on - first, we iterate through every batch of data we have in our dataset (in our case, to simplify matters, we have a batch-size of 1, and so we're just iterating through every element). For each batch, we run it through the model to get a predicted result, calculate the loss of that result relative to the actual result we know via our selected loss function, and then use our optimizer to take the number output by the loss function to update our gradients! The "forward pass" refers to when the model (even with its initially random weights) calculates the result of our input. The so-called "backward pass" refers to how the model calculates the gradients (based on the "graph" that your machine learning library of choice stores) and then updates.

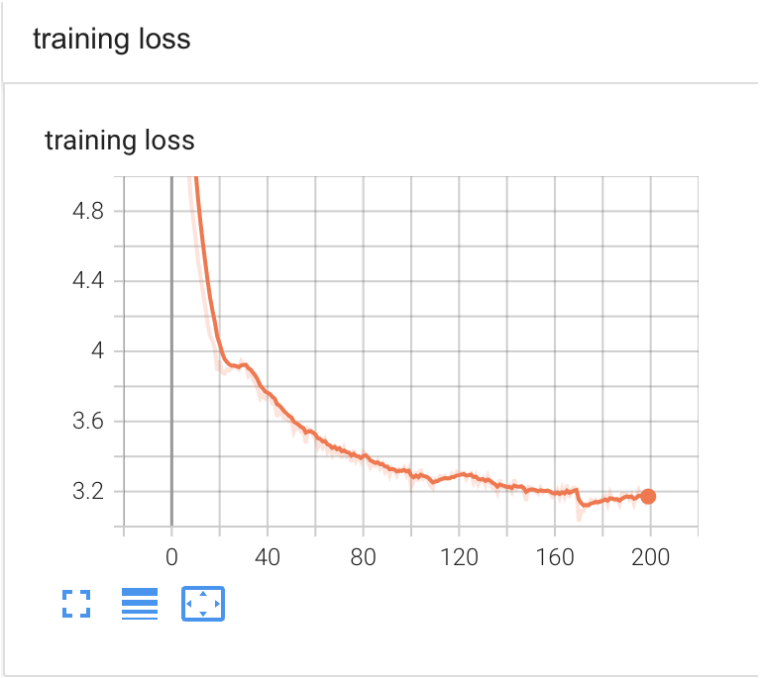


Figure 6: Training Loss

Notes: The fact that our test loss is so much higher than our training loss signifies our model probably overfits to the training data. We should consider reducing the complexity of the model in order to allow the model to generalize (ie to work for data the model has not seen before) better.

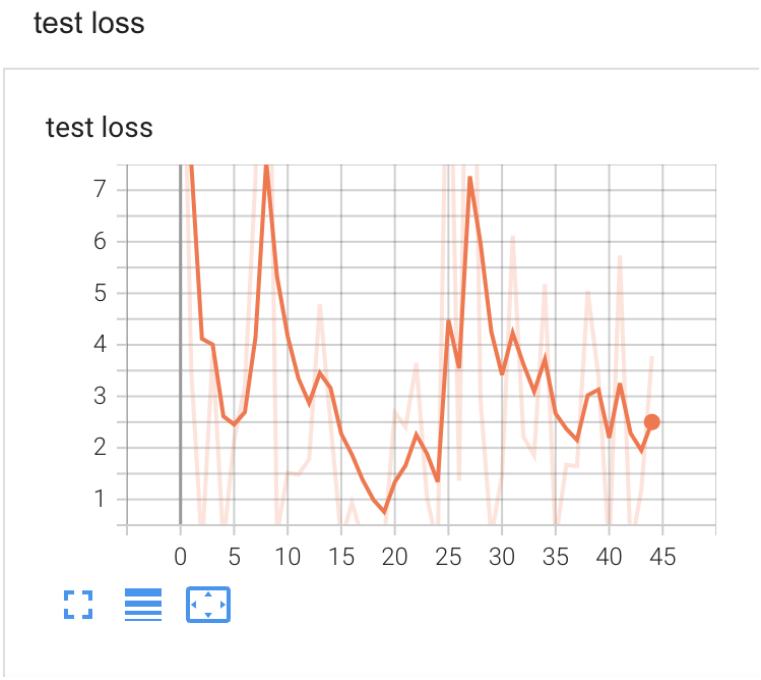


Figure 7: Test Loss

# 4 Convolutional Neural Networks

So, let's talk about convolutional neural networks for a moment now. The general definition is really quite simple - they simply are a type of model that rely on convolutional layers as opposed to dense/linear layers like we have been using.

However, how these convolutions work is a whole new can of worms. What is a convolution, anyway? A convolution (technically a cross-correlation, but ML people call it a convolution anyway), is the idea of multiplying a matrix of learned weighted parameters (potentially with learned bias terms) by input data, just like neural nets, but only on a subset of the data. Rather than using a whole matrix of weights, we use much smaller matrices, known as filters, multiply them by a few entries in the data, and get a single number as a result. We stitch these results together to get the output we want for the next layer. Additionally, one convolutional layer is not simply one filter applied across our input, but many filters.

Consider the case where I have one 3x3 filter traversing a grayscale image of say, 9x9 size. Consider that the center of the 3x3 filter starts in the upper left corner (specifically coordinate $(1,1)$). Know that our 3x3 filter does good work and outputs one number to summarize the information of that 3x3 square in the upper left corner. This will be the first number in our output. Afterwards, we move our 3x3 filter one pixel to the right, so the center will be at $(1,2)$, calculate that information, and then end up with an output matrix of 7x7, each entry $y[i][j]$ containing the information of all 9 pixels centered at $(i,j)$.

So, what is this magic formula that lets us accumulate information from the surrounding pixels into a single entry? It's nothing more than a simple dot product. The sum of the products of each corresponding entry in the filter matrix vs the input matrix, e.g.:

$$y[a][b] = \sum_{i=0}^{f.size()} \sum_{j=0}^{f[0].size()} x[i+a][j+b] * f[i][j]$$

Where $f$ refers to our filter matrix, $x$ refers to our entire input matrix, and $y$ refers to our output matrix. $f.size(), f[0].size()$ are usually the same, because we like to use square filters - in our example, both of

these are 3. $i, j$ represent the indices of our filter, while $a, b$ represent the indices of our output.

So what are the weights that our model updates? Our model updates the weights of the filter matrix $f$ so that the output image matches something we want.

Now, notice something interesting about this layer? It tends to decrease the dimensions of the input size. E.g., if we simply apply this directly on the image, the edges tend to be a little chopped off (because notice that they are only ever included as edges of our matrix, not as centers). We often want images passed between layers to not have this little drop off, and so we often pad our inputs with 0s so that the dimensions remain consistent throughout the running of our program.

Another important thing to realize is that our 3x3 matrix effectively zoomed us out of our image. Each entry in our 7x7 output is a conglomeration of the neighboring 3x3 pixels. If we were to directly run another 3x3 filter across our image, the output would be of size 5x5, and each entry in this 5x5 would involve the data of 9 adjacent (albeit overlapping) 3x3 kernels! If you draw it out, you would see that each entry in our 5x5 is effectively based on data from within 5x5 grid. This is a critical idea of most GANs - this zooming out gradually lets us learn more and more complex features.

But what if we want to zoom out a little faster? Going one by one with 3x3 filters or 5x5 filters might seem a little slow. Well, in that case we can use something called the stride - by default, we have a stride of 1, as denoted in our little toy example above. But we can lengthen our stride to cover larger amounts of data more quickly! We're effectively reducing overlap, until we reach a stride of 3x3, in which case we are perfectly covering the data once, and larger strides beyond that will start skipping pixels and losing data. Generally, I think strides of 2, or maybe at most 3 will suffice for downsampling purposes. You cover ground proportional to the square of the stride, so having a stride of 2 will leave you an output layer with only 1/4 of the number of entries as the input matrix. That reduces your size pretty quickly, for most all images, and you do want to give your model enough time to learn features at different levels.

The above describes a single convolutional filter applied on a grayscale (1-channel) image. Now, to clarify the terms filter and kernel - the kernel in the above case happens to be identical to the filter. However, in the case of a multi-channel input, the kernel refers to the square matrix of pixels that are being operated on at that time on that channel in particular, while the filter refers to the collection of kernels as a whole. I highly recommend drawing all of this out on a whiteboard (which I found very necessary, otherwise, this wall of text would be entirely unintelligible). Also, a note on dimensions - X by Y by Z - X, Y are the length, width, and depth, as expected.

Channels, as you might recall, are simply RGB channels of red, green, and blue, in the input image. Each filter operates on all three channels simultaneously. Note, however, that the dimensionality of the filter increases in the case of three channels. Rather than moving around a 3x3x1 square around the image, we are actually moving around a 3x3x3 cube around the image, taking the dot product at each entry, and then agglomerating that information into one entry. So, our for example, if our grayscale image were converted into a 9x9x3 (channel-last dimension) image, and our filter scaled up to a 3x3x3 cube, the output would still be a 7x7x1 flat square, not a 7x7x3 rectangular prism! Note that this also implies that our filter has 27 trainable parameters as opposed to 9, though there our kernel is still a singular 3x3 square moving around on each channel.

So, we have a single filter, running across our input image (remember that our image is grayscale and so only has one channel so we don't have to worry about those quite yet) looking for if this one feature exists. Why don't we increase the number of features we search for? A filter can really be thought of as a filter on a pixel scale, and were even crafted by hand in the olden days. For example, the following vertical filter could be used to detect where in the image a vertical line appears.

What about horizontal lines? Diagonal lines? We need to increase the number of features we can detect in our image so we achieve a better representation of our image. Thus, we simply add more filters to our model. More layers proves particularly powerful because they are operating on higher-level features - ie, let's say I have a diagonal line filter, and a straight line filter on the base input layer. Then, on the next layer, I might learn a filter that can effectively detect if there are three diagonal lines in these locations in the image, I actually have a triangle. Or, if I have three straight lines adjacent to each other, I have a long straight line detector. Then, at an even higher level, if I have a triangle next to a long straight line,

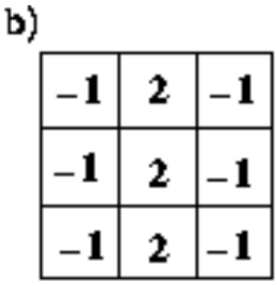


Figure 8: Shows vertical line filter

I actually have a yield sign, and this filter will search the image for the locations of yield signs, which you can imagine would be useful in say, a self-driving computer vision program. It might also become apparent that as our filters cover a wider and wider scale, we need more of them - ie there are only so many 3x3x1 squares possible ($256^9$, to be precise), but there are many many more different 5 x 5 x 1 squares ($256^{25}$). And so on and so forth. There are also more types of traffic signs (larger features) than there are say, lane markers (smaller features).

With this example, we also have a better example of what that one 7x7x1 output layer actually meant - it could have been detecting if there was a diagonal line in the 3x3x3 cube of pixels it was operating on. You could think of it as being binary, like, 1 for diagonal line found, 0 otherwise, but oftentimes our weights learned are not so black and white, but rather more gradiented, but it's helpful to think of it this way when learning it.

Now, let's look into some other types of layers. For example, what are transposed convolutional layers? Transposed convolutional layers (sometimes incorrectly referred to as deconvolutional layers), are primarily a way to increase the dimensions of an input layer. They work, effectively, the same as a regular convolutional layer, except that they fill in gaps between rows/columns with 0s to have more stuff to run the filter on, and therefore increase the resulting size of the output. In other words, when I have a stride of one, We can also stride when trying to go up in dimension, but that does increase the number of 0s present within a specific layer by a lot as well.

Ok, cool. What about max/average pooling layers? Max/average pooling layers are alternative ways to downsample as opposed to using strided convolutions. Another way to sort of aggregate data from surrounding pixels. Max pooling simply takes the maximum value of nine pixels (ie you could think of it as the strongest finding among a set of features in a given area), and then outputs just that as a result, ie that one max number. Average pooling is an entirely analagous idea, just instead of finding the maximum, we are finding the average of those 9 pixels.

One last topic to discuss - the number of filters we choose to create determines the number of output channels we have. But that output will still be a three-dimensional volume thing, and it's kind of hard to work with that, particularly if you want to be doing something like linear regression or classification.

While you could have an output matrix with just one filter, you could also choose to flatten the final layer of filters and their outputs into one long array. Then, we can even run this as an input through a traditional fully connected neural network and train it on that. There is a reason why this field is referred to as deep learning - the information that our model operates on is many layers deep, and can even be many models deep!

Let's do an example to test our understanding! This above figure shows different sized kernels operating on an input batch images. This diagram starts with a batch size of 8 $128x128$ pixel images that I assume to be grayscale (it's hard to tell based on the diagram thus far). For the batch of images (remember that when we batch images, we're effectively stacking them! Each layer only gets one input tensor),

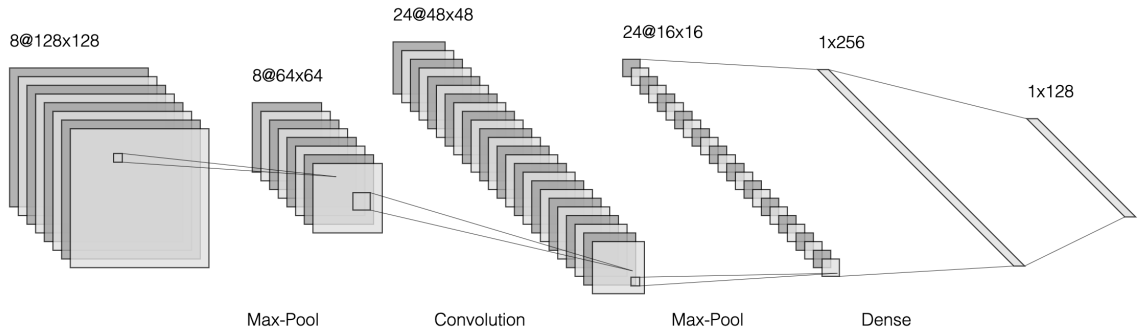


Figure 9: Example Model and its layers

we first max_pool (take the maximum of a given square region of 9 pixels), with maybe a kernel size of $3x3$, and a stride of two, with "Same" padding, so that the only loss in dimension size is from our stride parameter, and not the natural chopping off edge. We have 8 of these filters so that our output also has a depth of 8 (remember that we do not have one filter per batch element, we have one filter for the entire input tensor, and so to end up with an output depth of 8, we need 8 of these filters).

Then, this model has a convolutional layer with 24 filters and a massive a kernel applied to it. Given that strides must be integer, and 48 is not an integer multiple of 64, the kernel is probably of size 17x17 applied with "Valid padding" In general, the formula for determining output size of "Valid" padding is pretty simple and is as follows (remember Valid padding means that that kernel must stay within the bounds of the input, while Same padding adds Zeros along the edges to ensure the Valid cutoff doesn't occur):

$$Y = X - F + 1$$

Where Y is output dimension, X is input dimension, and F is the length of a side of the filter (we assume all inputs/outputs are square), though they certainly don't have to be.

Then, we, apply maybe another 24 3x3 kernel size filters with stride length = 3. to get our 24 16x16 results, which I believe they average and then flatten to get a 1 x 256 layer. This then passes through a fully-connected neural network to get 128 output nodes. This could be a classification problem with 16 classes (I'm assuming each input batch element gets their output read based on the index of the 128 pixels), or something else depending on what the loss function is.

# 5 Structures of Convolutional Neural Networks

The ideas discussed above and these new-fangled layers are easier to understand if we consider a real example, applying these layers on one of the most applications today, ie GANs.

So, first, let's introduce the architecture of a GAN. GANs stand for Generative Adversarial Networks, and operate on the principle of two neural nets competing with each other. They also adjust their learning rates so that (hopefully) neither model dominates the training.

The models are called the discriminator, and the generator. The discriminator is a binary classifier, and receives as input, two batches of images. One batch comes from the real samples, while the other batch comes from fake samples. The fake samples were generated by the generator, while the real samples, we have on hand.

Note that the weights of the disciminator should be frozen when training the generator. Why? Because otherwise if the discriminator updating its model while the generator were running, it would be even harder for it function.

So, let's talk about creating the Discriminator. This is the simpler model to create. First, we start by subclassing the PyTorch neural network module. We do this in a manner very similar to what we did last time, starting by overloading the constructor, adding in our layers, and then later defining the forward pass. So, for this binary classifier, we want an array of probabilities denoting how likely our discriminator thinks the current object is real. 1 for it is certain it is real, and 0 if fake, and most likely, somewhere in between. Sometimes, people threshold the probability and force it to give a firm answer - 0 or 1, e.g. 0.5 might be the most common threshold, but some use cases might want to set this threshold lower or higher depending on how painful a false classification is, for example (one can imagine this having impact on say, tumor classification - cancerous or benign) if they were actually using the classifier for its outputs, but we do not have such a requirement, because we need our losses to train our model.

to use a binary cross_entropy loss function.

What is a binary cross_entropy loss function anyway? Well, let's say we have an array of probabilities. This would presumably be the result of our machine learning model looking at some samples, and then spitting out some numbers representing how likely it thinks the predicted is truly correct. For example, consider the following table:

| Model Probability | Actual Result | Correct Probability | Loss |
|---|---|---|---|
| 0.92 | 1 | 0.92 | 21 |
| 0.3 | 0 | 0.7 | 123 |
| 0.4 | 1 | 0.4 | 123 |
| 0.12 | 0 | 0.12 | 123 |

Basically, since our model only outputs $P(x)$, in binary classification, sometimes we want $1 - P(x)$, or the probability of the other event. Therefore, we sometimes need to correct the probability depending on what the actual result was, e.g. $P(x) = 0.3 \implies P(!x) = 1 - P(x) = 0.7$. We want the "Correct probability" to be the probability that our model would in fact, produce the correct result. Then, we penalize the function based on how high/low the model predicted the example to be, and how far away it was from the actual results (1/0). We take the log of the differences to find the amount we want to adjust our gradient by. Then, since our answer will be negative, we multiply it by negative one, and backpropagate it as per usual (hence the name of negative log loss).

Now, let's discuss how to apply these loss functions in the context of our generator/discriminator. Firstly, the discriminator application of this loss function. Our discriminator gets input from two different directions - one via the fake data generated by the generator, and another from the real images we provide in batches. We therefore make two calls to the discriminator's loss function - one with a call like discriminator(real, 1s) to denote actual results, one with a call like discriminator(fakes, 0s) to denote the fake results.

Then, we call our discriminator's optimizer (ADAM) to take these loss functions and apply them across our batches. I would also like to make the clarification between epoch and step. Epoch refers to an entire

traversal of all of our training data, while a step can refer to a single batch of that training data. So, we can relate the two via the following simple formula:

$$epochs = \#steps/(training\_data\_size/batch\_size)$$

Now, how do we train our generator? Remember, our generator wants to increase the realism of each of our inputs, and we want the discriminator loss to tend towards 0.5 (ie it only has a random chance of guessing between real and fake). Given this, we apply BCE as follows:

$$generator(probability\_discriminator\_thinks\_generator\_images\_are\_real, ones)$$

In other words, this takes the probability generator images are real, logs it (returning negative outputs, which we multiply by -1) and penalizes it proportional to the distance from 1 - e.g. we are seeking to maximize chance of discriminator thinking that it's real (giving it a target array of 1s as opposed to a target array of 0s as we did the discriminator).

After calculating those gradients via backpropagation, we find the amount we can increment our parameters $\theta$ by, allowing us to update our model.

Then, we add in our Tensorboard scalar recording and a test loop - it's much simpler than with our other example of using a neural network to predict outcomes, because once a GAN is trained, we can basically discard the discriminator, because the inputs were just noise to begin with, and the generator's outputs should be entirely representative of what it was outputting during training.

Now, you may have noticed a pretty gaping hole throughout this process - we never recorded any data! This was partly because I was too bored/lazy to really worry about what data I wanted to include, so, yeah. But, at this point, I've written a data processing tutorial towards the top, and so I'm going to assume that you've taken your raw imagery and processed it into a nicer sized picture for our intents and purposes.

Then, our dataloading function only needs to work with the nice inputs, and we won't have to deal with the wild content that you get from directly scraping images/text from the web.

So, what follows will probably be more like notes and a summary of documentation, because loading data into one's application is actually very difficult, particularly when it comes to GPU (which my macbook pro doesn't even have, so I'm praying training won't be too bad).

PyTorch has DataLoaders, and Datasets. They are basically what their names make them out to be. Similar to how we subclassed our models, we can create custom datasets via subclassing.

We need to overwrite three methods - init (constructor), get_len(length of our dataset), and getitem(gets well, an item). Given that we have our preprocessing function handle bothersome things like jittering the image or flipping it, we simply need to load images. In particular, our GAN will want higher quality raw images. Additionally, given that we have finished our web scraper and preprocessing, we should be good to go.

So, let's see the results! It's not very good, especially given that I conducted all training on a MacBook Pro as opposed to like something with NVIDIA GPUs, but they'll simply have to do for now.

So, the reason why I only included two such samples is simply because my model experienced the phenomenon known as mode collapse, or in other words, because my GAN has found one set of images that are relatively good at fooling the discriminator and then continuously generates visually similar images, simply because this is relatively good at fooling it.

Given that the semester is starting soon, I decided to leave the results as they are for now, and then come back to finetune the model later. It was not for a lack of trying however, to fix my model, as you can tell by the screenshot below of my various generator loss functions :((. I hope your endeavors go better! There are a number of ways to resolve mode collapse - get a larger, more diverse dataset (mine was only 120 images), create a more complex model (possible with better hardware), or spend more time finetuning parameters.

U-nets are one example of a structure that is commonly used within a GAN, and our aforementioned convolutional layers.

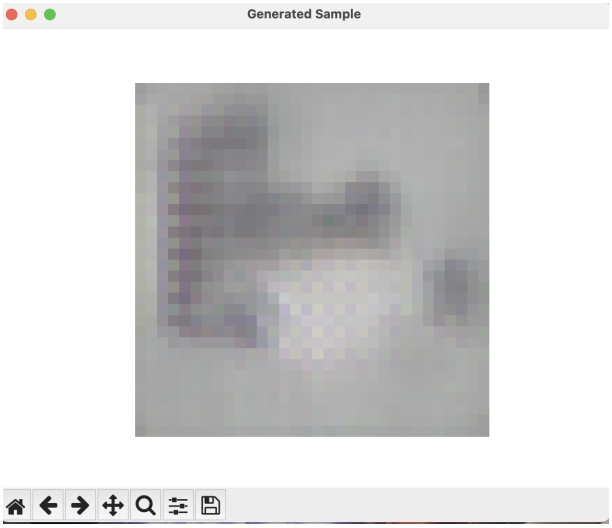


Figure 10: One sample generated by GAN.

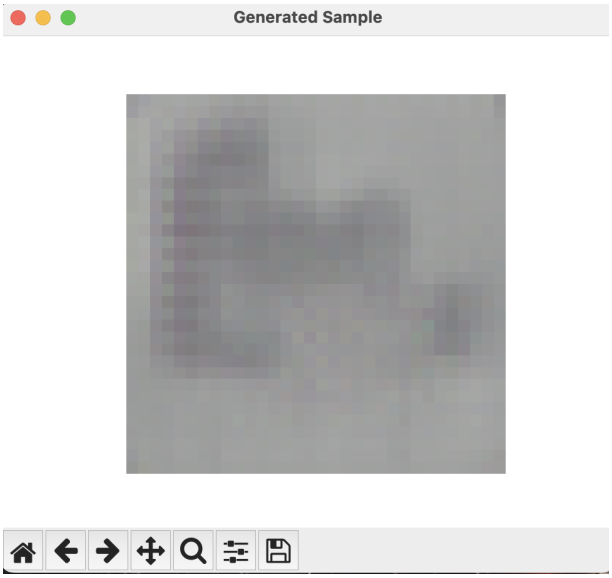


Figure 11: Another sample generated by GAN.

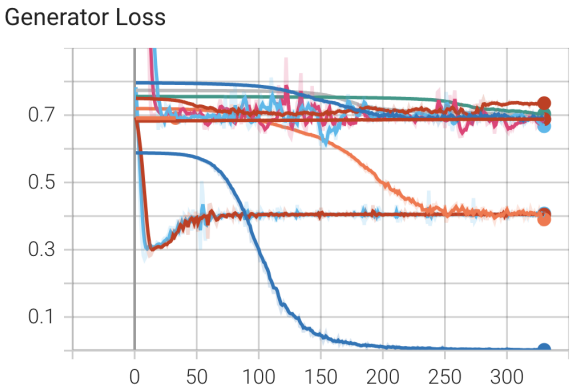


Figure 12: Evidence of my efforts at fine-tuning the model. Note how some of my changes broke the model (sending generator_loss to 0).

# 6 References

https://cs229.stanford.edu/main_notes.pdf
https://www.youtube.com/watch?v=jGwO_UgTS7I&ab_channel=StanfordOnline
https://homepages.inf.ed.ac.uk/rbf/HIPR2/linedet.htm
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
http://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf
https://archive.ics.uci.edu/dataset/10/automobile
https://bookdown.org/roback/bookdown-BeyondMLR/ch-beyondmost.html
https://github.com/alexlenail/NN-SVG
https://calcofi.org/data/oceanographic-data/
https://www.ebay.com/sch/i.html?_from=R40&_nkw=shoes&_sacat=0&_pgn=1