Lowestcase and uppestcase letters: Advances in derp learning

Dr. Tom Murphy VII Ph.D.

1 April 2021

1 Introduction

Have you ever been writing something on the internet and wanted to convey that you ARE FEELING ANGRY? Conversely, have you ever fired back a super quick dm and u wanted to make it clear that it was like super caʒ and so u didnt use ne capitals or punctuation dots except 4 that one place where u needed to use the international phonetic alphabet because u dont no how to write caʒ as in short for casual without it lol

If so, you made use of the fact that all letters have UPPERCASE VERSIONS (e.g. signifying ANGER) and lowercase versions (e.g. signifying u dont care lol). These dimensions have other uses, for example, it is polite to start a person's name with a capital letter to show that you took the time to regard their humanity (as it takes extra work to press the caps lock key, press the first letter of their name, and then press the caps lock key again to turn it off). In German, Nouns start with uppercase Letters, signifying Superiority over other grammatical Categories like Verbs and Adjectives. Lowercase letters can be used to conserve printer ink. Actually, I'm not sure that lowercase letters have any other uses, but let's just roll with it.

There's nothing wrong with this (despite the classical advice to use shift to *reduce* conflict [2]). But the thing is: What if I'm even MORE ANGRY THAN I WAS BEFORE? There are some standard sorts of typographic emphasis, like I can be **BOLD ANGRY** or ***BIG BOLD ITALIC UNDERLINE ANGRY*** or ***COMBINE A LOT OF THESE ANGERS***, each with its own nuances, depending on the cascading style sheet or LaTeX class file. To be even more casual than lowercase, u can learn 2 write like this, and shrink away and also ~~cross out ur words in shame in advance of them even being read~~, but there are few other options for de-emphasis. Plus, when I'm FEELING PRETTY ANGRY, TOM, how do I capitalize that already-capitalized T in order to show the proper reverence for your humanity?

This paper is about unshackling this dimension of human expression by introducing letterforms further along the uppercase and lowercase dimensions. Basically, we want to know what the upper*er*case version of uppercase T is, and a lower*er*case version of lowercase t is.

*Copyright © 2021 the Regents of the Wikiplia Foundation. Appears in SIGBOVIK 2021 with the OS2TypoLinegap of the Association for Computational Heresy; *IEEEEEE!* press, Verlag-Verlag volume no. 0x40-2A. 1 em

1.1 Induction

Today we're just concerned with English letters, of which there are only 26. To create an uppperercase and lowerercase alphabet by hand is O(52 pick up), which for a guy who likes drawing letters anyway and who alphabetized Star Wars for fun, is not much to ask. In fact I drew such alphabets in Figure 1 just now.

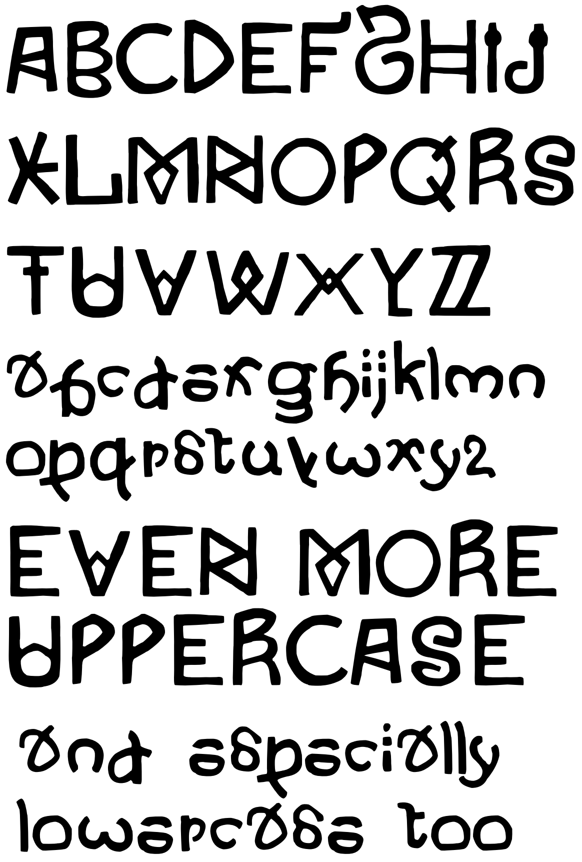


Figure 1: Probably someone already had this idea and did it before I was even born, thus taking the fun out of it for the rest of us, but here's a hand-made alphabet with "uppercase" and "lowerercase" letters. You can download this TrueType font from `tom7.org/lowercase`.

But, why do easy fun things by hand when you can build a complicated automatic solution which produces much worse results? Well, there is no good reason. I could claim that this allows us to automatically uppperercase any font, which is true, but the results are at best moderately letter-like lumps. In principle there are several other interesting things we can do, like apply the function over and over to approach the uppestcase and lowestcase letters. This sounds fun, but the results themselves are not going to impress. But the story of getting there may be interesting, and even as it turns out to be "derp learning," there will be opportunities for more good puns. So let's just roll with it!

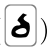
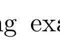
2 Capital A Artificial Intelligence

We want to machine-learn [7] two functions, make_lowercase and make_uppercase. Each takes a letterform and returns a letterform (we can choose how these are represented) and does the needful, e.g. make_lowercase($\boxed{\text{A}}$) should return $\boxed{\text{a}}$. In order to learn this function, we'll at least need a lot of examples to use as training data. A training example for make_lowercase is a letterform and its expected corresponding lowercase one. We can "easily" find a large amount of examples by using existing fonts, and pairing their $\boxed{\text{A}}$ with their $\boxed{\text{a}}$, and so on for all 26 letters, and symmetrically for make_uppercase.

However, if we only give uppercase letters to make_lowercase, it may very well learn how to generate the corresponding lowercase letter but be unable to do anything interesting for other letterforms. This is a problem because we want to use this function to see what e.g. make_lowercase($\boxed{\text{a}}$) is.

This is not (only) the problem of overfitting. An overfit model could work well on the letter $\boxed{\text{A}}$ from one font (because it has seen that font before) but fail on $\boxed{\text{A}}$ from a new font. The property that we want is that the learned function can also produce an interesting result on a shape it's never seen before, like $\boxed{\text{ʒ}}$. That is, it has generalized the idea of "how to make a shape lowercase," not simply "how to make a capital A shape lowercase."

The problem with this is that we don't have any training data other than existing fonts to tell us what the lowercase of some arbitrary shape should look like. Without examples of this form, the problem is unconstrained. make_lowercase could learn to generate empty output for anything it doesn't recognize as a capital letter, and still have perfect performance on the training and test set. It is hard to generate training data of this form (even by hand) as we don't have much idea *a priori* of what a lowerercase $\boxed{\text{a}}$ should look like (except for e.g. One Artist's Impression from Figure 1).

This brings us to the one decent idea in this paper (which by the way only sort of works, but let's just roll with it). We can at least express one characteristic property of the make_lowercase function that ought to be true even for letterforms we don't have examples of: It ought to be the inverse of make_uppercase. So, we train these two models in tandem. make_lowercase is fed training examples from the sample fonts like $\langle\boxed{\text{Q}},\boxed{\text{q}}\rangle$ etc. and make_uppercase gets $\langle\boxed{\text{e}},\boxed{\text{E}}\rangle$ etc. as expected. We also run the current version of make_uppercase on some letter-like shapes, which produces some other shape. For example, say that make_uppercase() outputs . We have no idea if this is good or not, so we don't update the model. However, we *do* provide the training example to ⟨ , ⟩ to the make_lowercase training queue and penalize *it* if it did not predict . In this way, whatever make_uppercase is doing, we ask make_lowercase to learn the inverse. We of course also simultaneously do the symmetric thing, using the output of make_lowercase to create training examples for make_uppercase (Figure 2).

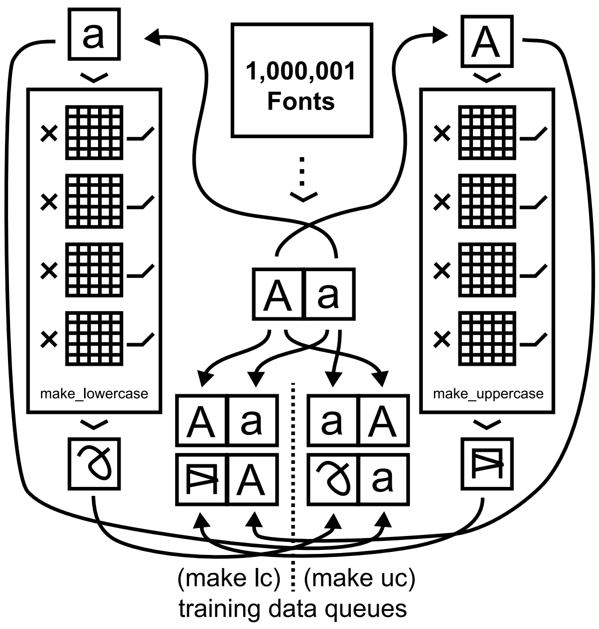


Figure 2: Simultaneously training the two models. This example illustrates how a pair of letterforms $\boxed{\text{A}}$ and $\boxed{\text{a}}$ from the same font becomes four training examples. The pair straightforwardly generates an example $\langle\boxed{\text{A}},\boxed{\text{a}}\rangle$ for the make_lowercase queue, and an example $\langle\boxed{\text{a}},\boxed{\text{A}}\rangle$ for the make_uppercase queue. Separately, we supply $\boxed{\text{a}}$ to the make_lowercase model, simply to get the current output (no model updates are performed). But this pair reversed becomes a training example ⟨ , $\boxed{\text{a}}$⟩ for the make_uppercase queue.

Because make_lowercase is getting training examples of uppercase/lowercase pairs from real fonts, it remains grounded on real letters. It is also free to generate new shapes for the open domain (outside $\boxed{\text{A}}$–$\boxed{\text{Z}}$). However, it is penalized if its behavior is not the inverse of whatever make_uppercase is currently doing. And since we do the symmetric thing for make_uppercase there is a (slow) feedback loop between the two models that keeps them from straying too far from the grounded examples. The idea is that this allows them to do some creative generalization outside their native domains, but in a way that still has some constraint.

In practice, we don't feed arbitrary shapes to the models. We just need something letter-like, and in fact we have a large collection of letter-like shapes among our existing fonts! We pass already-lowercase shapes to make_lowercase, in order to generate inversion examples for training make_uppercase. These shapes are clearly letter-like (they *are* letters) and are also of interest to us anyway, since we want to try to generate lowerercase and upperercase letters from the trained models.

3 1000001 Free Fonts

Sprechen of Fonts, I downloaded every font I could find on the whole internet. This was overkill. The resulting directory tree contained over 100,000 files, many of which were duplicates. Exact duplicates are easy to find, but since many of these files were the result of 30 years of community transmission, they had acquired various mutations. One of the first things I did was write software to automatically remove files that were essentially duplicates even if they weren't exactly the same bytes.

Next, my lord, do people have bad taste! And I say this as someone who made dozens of amateurish fonts [1] as a high school and college student and who is contributing several new questionable fonts as a result of this paper. The database is just filled with garbage that is unusable for this project: Fonts that are completely illegible, fonts that are missing most of their characters, fonts with millions of control points, Comic Sans MS, fonts where every glyph is a drawing of a train, fonts where everything is fine except that just the lowercase r has a width of `MAX_INT`, and so on. So I built a UI (Figure 3) for efficiently and mind-numbingly cleaning up the database by marking fonts as broken or suitable (and also categorizing them as serif, sans-serif, decorative, techno, etc., which classifications I never used). In doing this I noticed another extremely common problem, which was that many fonts had the same letter shapes for uppercase and lowercase letters. This would not do for the current application!

But why manually mark fonts with nearly the same upper- and lowercase letters, when you could build a complicated automatic solution? The first pass identified fonts whose letters were exactly the same, but this was only a small fraction of the problematic fonts. A common issue was that the lowercase characters were very slightly modified versions of the uppercase ones, often scaled and translated and then perhaps "optimized" during the font export.

So, for a given font, I want to reject it if for most pairs of cased letters $\boxed{\text{A}}$,$\boxed{\text{a}}$, $\boxed{\text{a}}$ is close to a linear transformation of $\boxed{\text{A}}$. This problem can probably be solved with math, but it didn't sound that fun. Instead I tried out a new tool, and it worked well enough that I've now added it to the permanent rotation: Black-box function optimizers.

Black-box optimization. If you have a function and want to find arguments that minimize its output, the most efficient techniques are generally those like gradient descent. (In fact, the backpropagation algorithm we use to train the neural network in Section 6 is gradient descent on the function that takes the model weights and produces an error value for each output node.) The problem with this is that you need to do some math to compute the derivative of the function, and anyway you need to deal with fiddly bits (Section 6.1) unless the function is convex and smooth, which it will not be. If you don't want to deal with that, and have a fast computer (and who doesn't?), black-box optimization algorithms are worth considering. Here, the interface[1] is just something like (C++):

```
double Minimize1D(
    const std::function<double(double)> &f,
    double lower_bound,
    double upper_bound,
    int iters);
```

which takes a function `f` of type `double` → `double`, finite bounds on the argument's value, the maximum number of times to call that function, and returns the argument it found that produced the minimal value. Not as fast as gradient descent, but in practice "if the function is kinda smooth" these optimizers produce excellent results! The chief selling point for me is that I don't need to think about anything except for writing the function that I want minimized, which I just express in normal code.

In this case, I render the letterform $\boxed{\text{A}}$ and then optimize a four argument function taking `xoff`, `yoff`, `xscale`, `yscale`. This function renders $\boxed{\text{a}}$ with those parameters, then just computes the difference in the two rendered bitmaps. This finds the best alignment of the two letterforms (under the linear transformation) in a few hundred milliseconds (Figure 4). If the disagreement is low as a function of the total pixels, then we say that the letters have the

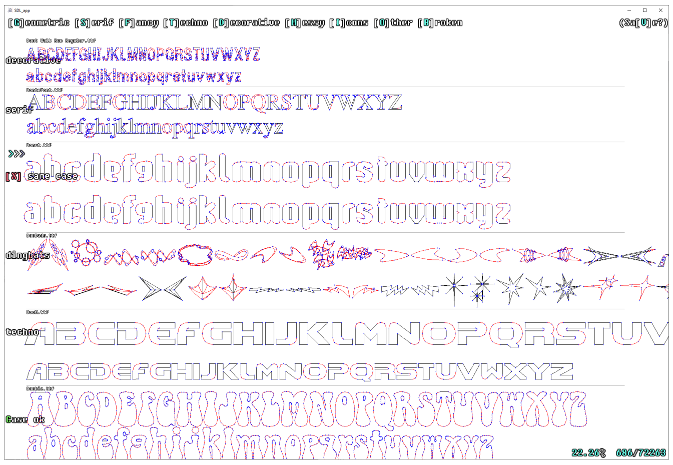


Figure 3: The interactive font data-cleaning UI. A seemingly endless series of fonts presents, with single keypresses putting the fonts into common categories such as (b)roken.

[1]Here a simplified wrapper around BiteOpt [21] in my `cc-lib` library. See `https://sourceforge.net/p/tom7misc/svn/HEAD/tree/trunk/cc-lib/opt/`.

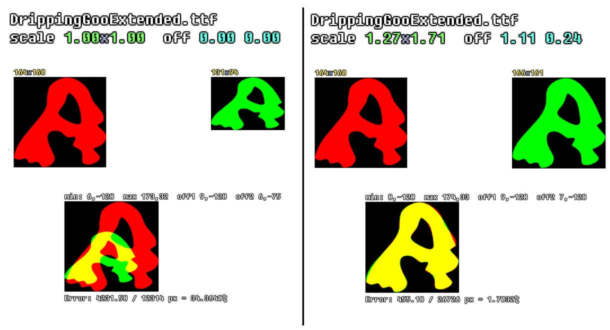

DrippingGooExtended.ttf
scale 1.00x1.00 off 0.00 0.00
164x160
131x94
min: 6,-128 max 173,32 off1 9,-128 off2 6,-75
Error: 4231.58 / 12314 px = 34.3640%
DrippingGooExtended.ttf
scale 1.27x1.71 off 1.11 0.24
164x160
166x161
min: 8,-128 max 174,33 off1 9,-128 off2 7,-128
Error: 455.18 / 26726 px = 1.7032%

Figure 4: Example alignment to reject the font `DrippingGooExtended`. At left, $\boxed{\text{A}}$ (red) and $\boxed{\text{a}}$ (green) rendered with the identity transform, and their alignment (35% difference) below. At right, the transform found by the black-box optimizer and the resulting alignment with 1.7% difference. Note that the shapes are still not an exact match (probably noise introduced in the font export process, which has to round the data to integers and might apply other non-linear transformations like curve simplification), but these are clearly not a useful pair for the current problem.

same case. If enough of them have the same case, we reject the font. I set the thresholds by looking at the P/R curve computed on random hand-labeled examples (Figure 5).

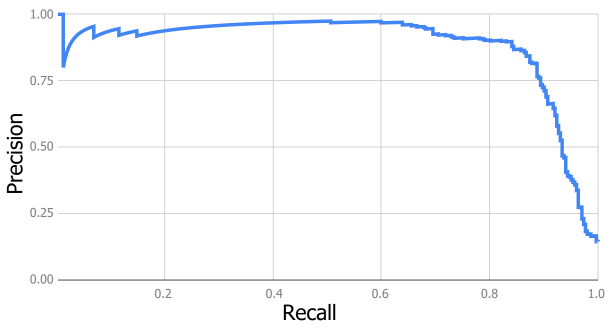


Figure 5: Precision–recall curve for automatically detecting fonts that have basically the same upper- and lowercase shapes. It's good! This is how you want 'em to look!

I labeled fonts using the UI until I had 10,000 that were clean enough for a training set and passed the same-case heuristic.

4 The simplest thing that might work

Before getting fancy (which we we will) it's good engineering hygiene to try the simplest thing that might just work (it doesn't). Fonts are represented as vector data (lines and quadratic Bézier curves). Can we just train a network that takes these lines and curves as input and predicts the lower- or uppercased letter in the same format? (No.)

We'll at least put the data in a somewhat normalized form. The neural network will take a fixed number of inputs to a fixed number of outputs, so a simple approach is to decide on some maximum number control points per letter, and only try training on fonts whose letterforms fit within that budget. Letterforms can be made of multiple contours (e.g. a stacked $\boxed{\text{g}}$ typically has two holes in it, and $\boxed{\text{j}}$ has two disjoint parts). I found that most clean fonts had three or fewer contours, and when sorting them by descending length, basically all of them fit within 100, 25, and 16 endpoints for the three. So, I only train on fonts where all of the letters fit within this budget.[2]

Rather than try to work with both lines and Bézier curves, I normalize each contour to only contain Béziers, by turning a line segment into an equivalent Bézier with its control point at the midpoint. This frees us from having to distinguish the two types in the data. We also need each of the three contours to not be *too short*, so I fill out the fixed-size buffers by repeating the last point. This is not great but does have the correct meaning (bunch of useless zero-length edges). It has the property that any predicted data can be rendered and has a straightforward point-by-point error function (which might not be the case if we were predicting a dynamic number of points).

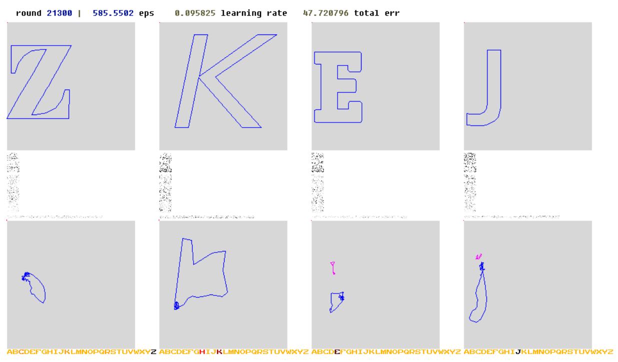
The network I trained has an input layer size of $570 = (100 + 25 + 16) \times 4 + 3 \times 2$ (one control point and one end point per Bézier curve), plus a starting point for each of the three contours. The output layer is the same size, plus 26 (see below). There are three hidden layers of size 308, 308, 360. The first layer is dense and the remainder are sparse, for just about 1 million total parameters. All layers are leaky rectified linear units (`x > 0 ? x : 0.1 * x`), which is fast to compute and better than sigmoids in the output since correct values will not just be 0 and 1. If you're taking notes, don't, as again this does not work well, and I don't know how people figure out what the right network size is anyway. I just made it up. You can give *me* your notes.

Bonus outputs. The output includes the predicted shape, and also 26 individual predictors for each of the 26 letters. So a training example is actually like $\boxed{\text{C}} \to \boxed{\text{c}}$ $[0, 0, 1, 0, 0, \ldots, 0]$, with the 1 in the third place because C is the third letter. We don't need these outputs for the problem itself (e.g. to lowercase new letter shapes), but there are several ideas behind this. First, the lowercasing function we're trying to learn does depend on the letter of the alphabet being lowercased (in an extreme case, consider the lowercase-L $\boxed{\text{l}}$ and the uppercase-i $\boxed{\text{I}}$, which look the same in many fonts but have different lowercase letterforms). By asking the network to learn this as well (it is penalized when it gets the prediction wrong), it must learn features that allow it to distinguish different letters, and

[2] It would not be a good idea to reject only the letters that don't fit, because it might result in the network being trained on more $\boxed{\text{l}}$s (tends to be simple) than $\boxed{\text{g}}$s (tends to be complex).

those features are available for use by outputs we *do* care about. This is an easy way to coax it to learn features that I know are meaningful without having to actually engineer feature extractors by hand (or first train separate models, etc.). Similarly, I could have asked it to predict whether the font is italic, serif, the character's width, or anything else I have on hand. Perhaps the most useful thing is that it's very clear what the right answer is, so it gives me an easy way to see if the network is learning anything *at all*. (It does.) Finally, we can do some silly stuff with these; see Section 7.

I trained the network using a home-grown (why??) GPU-based package that I wrote for *Red i removal with artificial retina networks* [16]—an example of "lowercase i artificial intelligence"—and have improved as I repurposed it for other projects, such as *Color- and piece-blind chess* [17]. It is "tried and true" in the sense that "every time I tried using it, I truly wanted to throw my computer out the window, and retire to a hermitage in the glade whenceforth I shall nevermore be haunted by a model which has overnight become a sea of infs and NaNs."



```
round 21300 |  585.5502 eps    0.095825 learning rate   47.720796 total err
```

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Figure 6: Screenshot of `train.exe` running on the first vector-based version of the problem. Shown is the make_lowercase model's output (bottom) on four shapes from four fonts (top). Some dust in between is the activation of the network's layers. At the very bottom, the 26 predictions for "what letter is this?". The output for $\boxed{\text{j}}$ is not too bad; you can see the distinct dot (a separate contour) and it sort of looks like a $\boxed{\text{j}}$. The $\boxed{\text{e}}$ also has two pieces as expected but is otherwise garbage. The model is unsure whether the second input is an H or a K, and has predicted a shape sort of ambiguously between those two. The $\boxed{\text{z}}$ is also an embarrassment.

I was so confident that this wouldn't work that I only trained a make_lowercase model and didn't even worry about the more complicated simultaneous training setup yet. I ran this for about 22,000 rounds, some 90 million training examples. Indeed it does not work (Figure 6). It is not a total catastrophe, though. We can tell from the 26 bonus outputs that the model can clearly recognize letters (though perhaps just by memorization). Some of the shapes it generates are along the right lines. (*Along the right lines*, get it??) I did not feel ANGRY at these results because I expected it to not really work. Still, it "has output" and so it can be used to generate a font. I made every glyph in Comic Sans MS [4] lowercase using the model (with the exception of the $\boxed{\%}$ character, which has too many contours—*five*!). Mostly this model produces small, non-confident scrawls, like little grains of sand, so this font is called **Comic Sands** (Figure 7). The TrueType version can be downloaded from my website and installed for your desktop publishing needs.[3]

4.1 Just try making it more complicated!

This problem of predicting the vector shape directly is a lot to ask of a neural network, at least set up this way. One thing that did not sit well with me is that the network could in principle generate a perfect-looking result, but because it didn't have the points in the expected order, it would be penalized. This makes it harder to learn, and more prone to overfitting.[4] This was one case where my questionable reflex to make things more complicated did pay off!

First, I reduced the number of points in the input and output. Reducing the dimension of the function being learned generally makes learning a lot faster. This had the side-effect of reducing the number of eligible fonts (by about half), and by nature these fonts are simpler shapes. These effects alone could be responsible for the improved performance of this second try.

I also output each contour's points in a normalized order, starting from the point closest to the origin. This removes one needless degree of freedom.[5]

Aside from the changes in the input (now 254 nodes) and output (280), this second version has three sparse hidden layers of size 508, 508, and 560 nodes; the first two are dense and the latter sparse. The final model after some pruning had 609k parameters.

As this was training, I worked on another improvement. Ideally we would compute the difference between the predicted shape and the expected shape, regardless of how they're drawn. Aside from being a bit computationally challenging, this won't really work because we need to attribute error directly to each output in order to actually update the model in training. I spent a valuable vacation day writing a routine to compute the best alignment of points between the actual and expected outputs (Figure 8). Aside from being harder than it looked, my alignment code

[3]Font downloads are available at `http://tom7.org/lowercase/`.

[4]For example, imagine if the database contains two versions of Helvetica that just have their points in a different order—which is very likely the case btw—the model will have to learn how to distinguish between these, but using information we just don't care about.

[5]We can see (well, it's not pictured since I have far exceeded a reasonable number of figures in this paper, but *I* can see) how this manifests in the biases on the output layer, which are a proxy for the "average prediction". In the first model, because of the unstructured order, these are mostly near 0.5 (center of the character) or 0.0 (degenerate, unused contours). In this new model, the distribution of biases is much more flat; it can learn that "the first point tends to be near 0.25,0.25" and "the seventh point tends to be near 0.64,0.3."

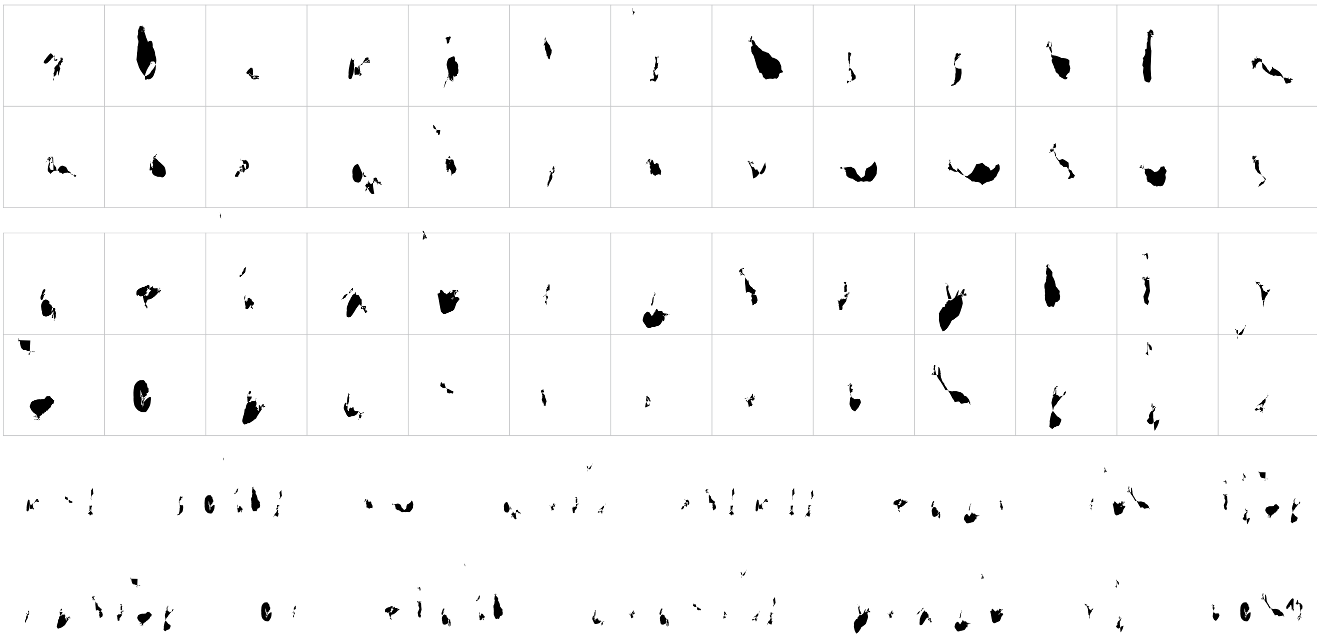


Figure 7: Type specimen for the generated font **Comic Sands**. This is the hateful Comic Sans MS run through an early vector-based lowercasing model (Section 4). At top are Comic Sans's letterforms $\boxed{A}$–$\boxed{Z}$ run through the model and so "made lowercase" (it's obviously garbage). Next are $\boxed{a}$–$\boxed{z}$, made even more lowercase. Also rubbish. At the bottom are the illegible pangrams "Dr. Jock, TV Quiz Ph.D., bags few lynx" and "Sphinx of black quartz, judge my vow!" Although the output barely resembles letters, it does have a certain wispy Rorschach aesthetic, like a collection of delicate moths pinned to paperboard, that one could consider framing or publishing in the proceedings of SIGBOVIK 2021. It is certainly an improvement on the original font.

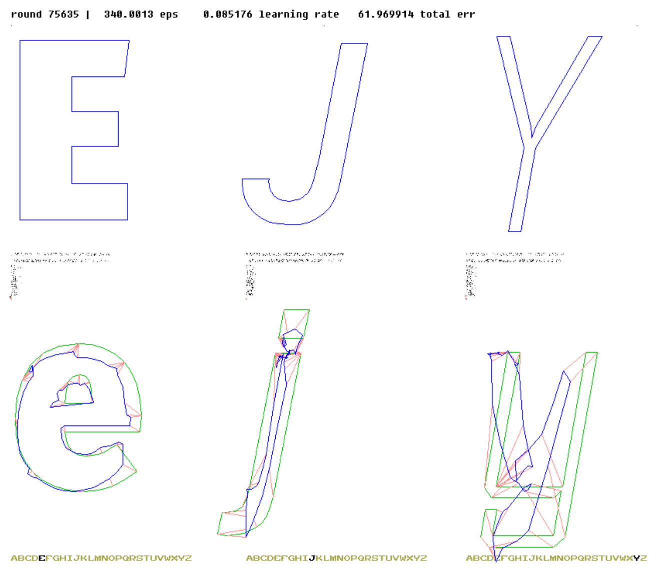
ended up being pretty slow relative to the rest of training, even worse since it ran on the CPU instead of GPU, which reduced the training speed by 50%. I let it run for 80,000 rounds, some 331 million training examples, but eventually got bored of waiting on this approach that was slow to train and seemed like a complicated version of an bad, oversimplified approach. So, I control-C'd that thing and threw this whole endeavor in the trash! But I must have confused the Recycle Bin icon with the fairly complicated export-to-TrueType Font process that I built, because I ran the model on the venerable Futura [19] font and generated **Futurda** (Figure 9).

5 SDFs

Don't give up! The fixed-size input/output of neural networks is better suited to something like an array of pixels, and fonts can of course be represented this way as well. To stay in the realm of what my desktop computer with a single GeForce 1080 can do, I wanted to keep the number of inputs and outputs pretty small. There's already an excellent technique for representing font data as compact bitmaps, which comes from computer graphics, called Signed Distance Fields (SDFs) [10]. In a standard rasterization of a font, each pixel of a bitmap contains 1 or 0, or perhaps an anti-aliased value in-between. In an SDF, the bitmap instead contains the distance to the nearest edge, and is signed (e.g. values inside the shape are > 0, value outside are < 0). Actually in practice we offset and saturate the values so that they are all in $[0, 1]$ (or bytes in $[0, 255]$), with some nonzero "on-edge value" (say, 0.5) standing for "distance 0". In order to display the font at the size of your choice, you then resize the SDF image with bilinear interpolation, and then simply threshold the image. This works surprisingly well (Figure 10).

SDFs seem well-suited for machine learning. They contain more information per pixel than a plain bitmap, so we can use a smaller input and output size. On the input side, extremal pixels that would almost never be set in a bitmap still have significant information (distance to the character). The error function is just pixel-by-pixel difference. The rendering of the output is inherently tolerant of some noise because of the sampling and thresholding. So, this seemed like it might work really well! (It doesn't work that well.)

I computed some stats on the font database, and determined the following parameters for the fixed-size SDFs we train on. The images are 36×36 pixels. The character box is placed such that there are 2 pixels of top padding, and 9 pixels of left and bottom padding. The character box is only "nominal" in the sense that the font's contours can exceed its bounds, and this is completely normal for a letter like $\boxed{j}$ (which goes below the baseline and often hangs to the left of the origin as well). I used an "on-edge value" of 0.862 (because much more of the SDF is outside the letter than inside) and the distance is scaled as 0.059



```
round 75635 |  340.0013 eps    0.085176 learning rate   61.969914 total err
```

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Figure 8: Screenshot (somewhat compacted) of training from near the final round of the vector model's training, illustrating the permissive loss function that finds the best alignment. At the bottom are the predicted lowercase shapes (blue), also shown with their expected shape (green). We require each point to be mapped (red) to a point from the expected contour in a monotonic order (but several can be mapped to the same one), so that we can attribute error to each point.

units per pixel (chosen so that pixels on the outer edge often have non-zero values). Compared to the first version, I was somewhat more permissive in what fonts I trained on, since there was no inherent limit to the number of contours or their complexity. I did exclude fonts whose rasterizations exceeded the bounds of the SDF, which is possible (very wide $\boxed{\text{W}}$ or low-descending $\boxed{\text{j}}$ perhaps) but rare.

6 The care and feeding of sparse matrices

Having committed to the representation, again it is "just" a matter of heating up the GPU to apply some linear and non-linear transforms. The initial network had an input size of $36 \times 36 = 1296$ for the SDF, and the output the same plus 26 bonus outputs (one for each letter, as before). I started with three hidden layers of 1296, 1296, and 2916 nodes, each sparse (80% of the weights are zero). Again, don't take notes. This one works a bit better than before, but still not impressive. The node references are assigned spatially (something like the 20% of the nodes on the previous layer that are closest to the next layer's node) but due to a bug the spatial locality is actually pretty strange. Every layer's transfer function is "leaky relu" again. It would definitely make sense to use convolutional layers for this problem, as features like serifs, lines, curves, and so on could appear throughout the input and output. I just haven't built support for that in my weird home-grown software, yet.

I also adapted my weird home-grown software to train the make_uppercase and make_lowercase models simultaneously. Two models fit easily in GPU memory, with plenty of space for a stream of training data (one training instance is only about 10kb). The only challenging thing is arranging for them to feed each other generated "inversion" examples (Figure 2), but this is just a matter of programming, thank god. I should remember to do projects that are mostly a matter of programming. Each round, 25% of the batch consists of inverted examples from the symmetric model's output from a recent round. Training happens asynchronously, but I make sure that one model is not allowed to get more than 2 rounds ahead of the other, because I want this feedback loop to be somewhat tight.

So I did that and let it run for a month. Actually I had to start over several times with different parameters and initialization weights because it would get stuck (Figure 11) right away or as soon as I looked away from the computer. I prayed to the dark wizard of hyperparameter tuning until he smiled upon my initial conditions, knowing that somewhere he was adding another tick-mark next to my name in a tidy but ultimately terrifying Moleskine notebook that he bought on a whim in the Norman Y. Mineta San Jose International Airport on a business trip, and still feels was overpriced for what it is.

6.1 Fiddly bits

The training error over time appears in Figure 13. It looks like the ones I have seen in machine learning papers, although I don't like to read other people's papers because it just seems like spoilers, and reading is the opposite of writing! There are several noticeable events in the curve, which came from me fiddling with the parameters or network as it ran. Here are some of the things I did:

Vacuuming and culling. Sometimes a node will just be dead (basically never activates) or an edge weight will be nearly zero. In these cases an equivalent, tidier network can be made by dropping the node or edge. Periodically I would perform these processes, sometimes feeling particularly choppy and removing like 10% of the parameters at a time. If these parameters are truly useless with no hope of recovery, we simply get faster training because there's less work to do. Speed is exhilarating!

Widening. The opposite thing is to introduce new nodes. Adding nodes to hidden layers is pretty easy. The thing that worked best for me is to increase the size of the layer by 10–15%, where each new node has random incoming weights and bias 0. Then for each node on the next layer, I add edges to some subset of these new nodes (again generally 10% of them) with weight 0. Since this weight is zero, the network computes the same function, but has new gradients to explore (in practice, it then experiences some shock after a few training rounds, but then quickly fine-tunes this away). More parameters means slower training, but also more potential to learn interesting functions, or overfit! Danger is exciting!

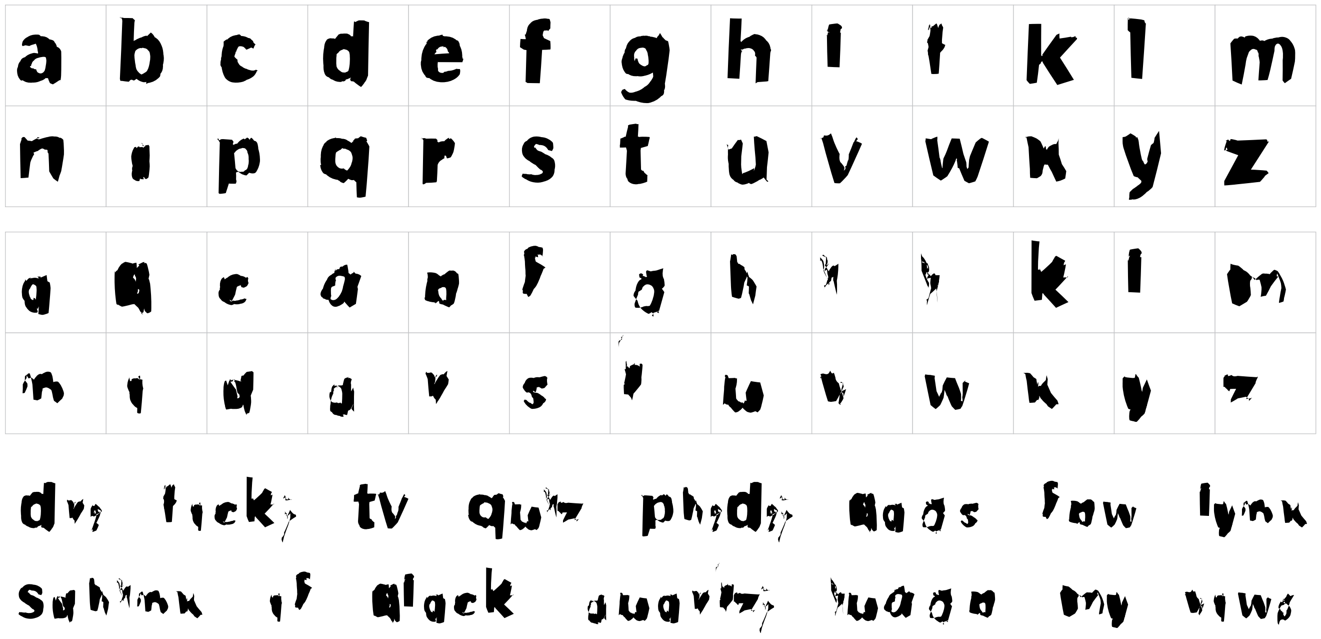


Figure 9: Type specimen for the generated font **Futurda**. This is the classic font Futura, run through the final, improved vector-based model (Section 4.1) to make each letter lowercase. The letterforms $\boxed{\text{A}}$–$\boxed{\text{Z}}$ (top) become quite readable lowercase versions. The extra-lowercase $\boxed{\text{a}}$–$\boxed{\text{z}}$ are also almost legible, but are mostly just scaled-down and screwed up versions of the lowercase letterforms. Could definitely imagine this appearing in the "distressed fonts" category of a 10001 Free TrueType Fonts CD-ROM in the 1990s, though.

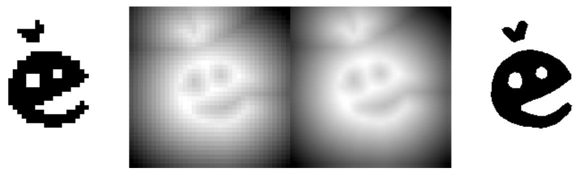


Figure 10: The signed distance function representation of a letterform. At the very left, a 36 × 36 pixel rasterization of the character without anti-aliasing, for comparison. Any scaling of this will have chunky pixel artifacts. Next, a 36 × 36 pixel SDF of same. Third, simply scaling that 36×36 image to 180×180 pixels with bilinear sampling. Finally, that image thresholded to produce a 180 × 180 pixel rasterization, which is far superior despite being derived from a 36 × 36 pixel image. Typically this process is performed at an even higher scale and then downsampled to produce an anti-aliased image.

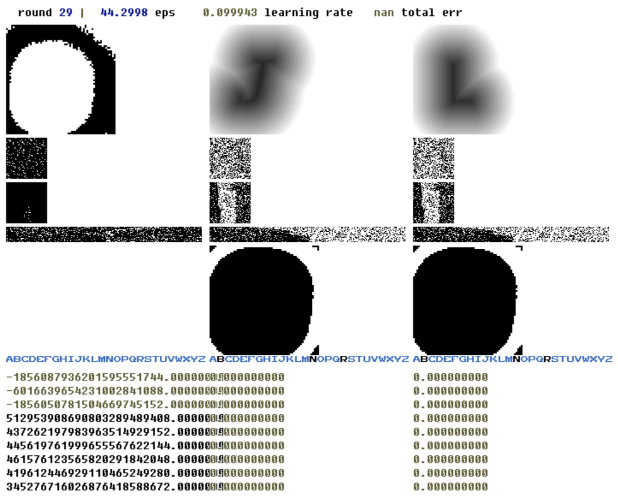
Deepening. It's also possible to add layers once the network is trained. This can be done anywhere, but I liked doing it on the output layer because this gets the most direct feedback from the training examples, and so it updates quickly and changes there are easy to understand. If you append the identity matrix (new layer is the same size as the previous; each node has weight 1.0 to its corresponding node and 0.0 elsewhere) then this network computes the same function but has new gradients to explore. Adding a layer did seem to help unlock a new training regime (Figure 13); subjectively it also reduced some weird artifacts in the SDFs that the model used to predict (makes sense; this most natural thing for this layer to do is learn how to predict a "correction" from the old prediction, for example by smoothing/sharpening it). This seems to be borne out by the weights, which are also fun to look at (Figure 14). All problems in computer science can be solved by an additional layer of indirection!

Generating features. On the other side, randomly sampling pixels from the input SDF does work, but I superstitiously believed that it might be better to have more spatially meaningful features. I wrote a program to generate a bunch of random simple features (one line/blob with positive weights, one line/blob with negative weights). It then chooses a set of them that are both *good* (maximum standard deviation on a sample of training data) and *different from one another* (redundant or even partially redundant features are less valuable). It was nice to satisfy my superstition, and the dark wizard of superstitious fiddling with neural networks in the hope that they do the thing was pleased as well. The features are at least handsome (Figure 12). Creativity is enriching!

I presume these are all standard things that neural people do, but they do better and smarter versions of them because they are willing to read other people's papers instead of trying to figure things out from scratch all the time. But you gotta occupy yourself somehow while it crunches for a month.

For completeness, some other innovations that I feel are worth mentioning:

Making the GPU code faster has really high value (could save weeks of waiting). Since I am using OpenCL (whoa,


```
round 29 |  44.2998 eps    0.099943 learning rate   nan total err
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ ABCDEFGHIJKLMNOPQRSTUVWXYZ ABCDEFGHIJKLMNOPQRSTUVWXYZ
-1856087936201595551744.0000000000000000                   0.000000000
-6016639654231002841088.0000000000000000                   0.000000000
-1856050781504669745152.0000000000000000                   0.000000000
5129539086908032894894 08.000000000000000                  0.000000000
4372621979839635149291 52.000000000000000                  0.000000000
4456197619996555676221 44.000000000000000                  0.000000000
4615761235658202918420 48.000000000000000                  0.000000000
4196124469291104652492 80.000000000000000                  0.000000000
3452767160268764185886 72.000000000000000                  0.000000000
```

Figure 11: Divergent training after only 29 rounds. We have NaN total error (hard to say if that's good or bad?). The example in column one is an inversion example generated by the make_uppercase model, which is why it also looks like the Tunguska event, just of the opposite sign. The other two are regular inputs, whose predicted outputs are black holes. Start over!

yeah, stop me right there, I know) I found a good technique was to generate different OpenCL code with constants baked for each layer (for example their size and `indices_per_node`); this allows the compiler to use faster tricks in inner loops for e.g. multiplication by a compile-time constant instead of depending on an argument or data. I have different routines for sparse and dense layers. It might even make sense to recompile the kernels for other parameters that change over the lifetime of training, like the learning rate. The `fma` instruction (so named for the physical law $F = MA$) is a bit faster than `potential += w * v`, and I guess the compiler can't do this itself because of IEEE horrors. But like, who cares? In my opinion you should be able to put it in "fast machine learning mode" where it readily makes precision errors, with a command-line option like `--fml`. With all the tweaking, the easiest win was to use the `restrict` keyword on arguments to tell the compiler that the input cannot alias the output, for example; this presumably helps it schedule instructions better.

Various things in training run in parallel threads (e.g. processing fonts, but also moving data to the GPU, backpropagation for each example, etc.). For a long time I had just been explicitly setting parallelism using superstitious constants. For this project I finally just wrote something that would automatically and empirically determine the number of threads that yielded the highest throughput, and persisted that information across program starts. This was a good idea and enters the permanent rotation.

The actual error on the predicted SDFs is pretty low; for the make_lowercase model it is around 31.3, which is like if 2.4% of the pixels were (completely) wrong, but the rest is exactly correct. In reality, of course, the error is distributed

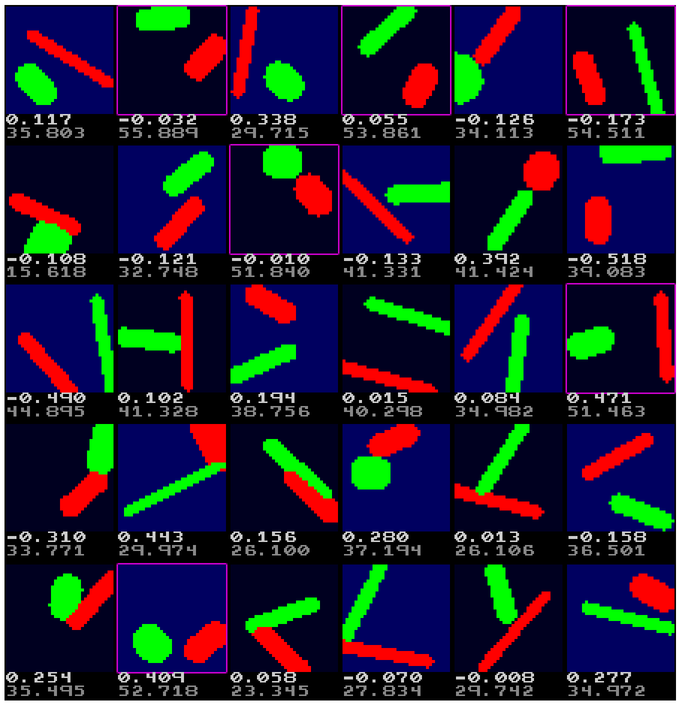


Figure 12: Some randomly-generated features with the selected ones outlined in magenta, mostly shown here for aesthetic reasons. Savvy Twitter user `@iotathisworld` sees this as "the classic question: machine vision classification or 90s roller rink carpet pattern?" to which I deflect: "Sorry, it's actually modern day Port Authority bus upholstery or Gram stain of same!" (But actually machine vision classification was basically correct.)

throughout the pixels, and some errors are a lot more important than others. Particularly, near the threshold value, a pixel goes from from being considered "in the letter" to "outside" with tiny changes in its value. Changes to a pixel with a value near 0.0 or 1.0 usually doesn't affect the output shape at all, in contrast. So one thing I did was map the loss function (comparing expected pixel value to actual) to "stretch out" the region near the threshold, increasing the penalty (basically, the derivative) in that region and decreasing it elsewhere. Looking at the code again right now, I realize that I only applied this to the first row of the SDF (`idx < SDF_SIZE` instead of `SDF_SIZE * SDF_SIZE`), so that was dumb AND MAKES ME **ANGRY**. I will say in my defense that at least I felt disappointed at the time that it didn't seem to make a difference! (The dark wizard of superstitious fiddling nods sagely.)

Ultimately, each of the two models was trained for over 2 million rounds, which corresponds to 510 million training examples. Each model is about 24 megabytes.

6.2 Uppercase and Lowercase fonts

Now that we have these expensive models, we can use them to make arbitrary letterforms uppercase or lowercase. The output is readily rasterized (using the standard threshold-

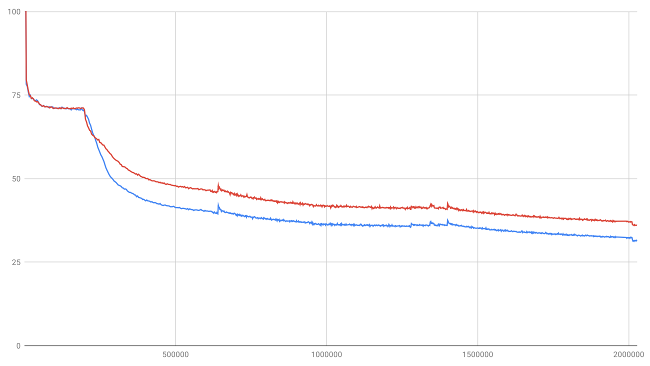


Figure 13: The training error for the SDF models. The red curve is the make_uppercase model, which generally has a higher error rate (perhaps simply because uppercase letters usually have more pixels set) and blue is make_lowercase. The first few rounds have error that's off the charts, well above 100. The most dramatic event is around round 200,000, where I reduced the weight decay factor to 0.999995 (from 0.9995). I guess you just need more nines to be more reliable. There are some other visible peaks, which occur when I do things like remove nodes with very low weights or which are almost never activated (Section 6.1). These momentarily increase error but it is easily fine-tuned away (e.g. by learning new biases). The peak at around 1.4M rounds is when I added a new layer to the end of the model, which does seem to create a new training regime (clear downward slope now); but this also significantly increases the training cost per round. Even after 2,000,000 rounds, the network is still apparently improving, but at a speed of about 1 pixel loss per several weeks. Eventually the extremely strict SIGBOVIK deadlines mean you just have to call it done.

ing approach for SDFs) but we'd actually like to have vector representations so that we can download the TTF files and clog up our fonts menu forever.

6.3 Tracing

Automatically tracing bitmaps into vector form is no doubt a solved problem, but I chose not to look at spoilers. Since we actually have a signed distance field, we can build a tracing routine directly off of that. The approach I took consists of three steps. First, I generate a bitmap of the SDF (at its native size) using the threshold. I separate the image into a nested tree of connected components in this pixel space; each component knows its single parent and whether it is "land" (inside the shape) or "sea". Characters like $\boxed{e}$ need internal cutouts, which are represented by a different winding order (clockwise or counter-clockwise) for the contour. Some of the tricky cases in computing this tree structure are given in Figure 15. Once I have this tree structure, I trace each pixel mass recursively (Figure 16). I find a pixel on the edge, and then walk around that edge

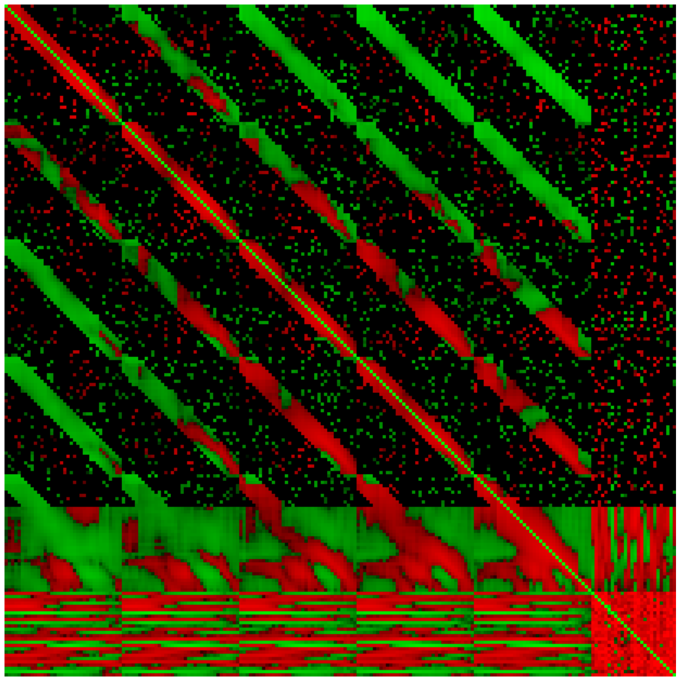


Figure 14: The bottom-right corner of the weight matrix for the final layer of the network. This layer was added to the network after 1.36 million rounds, initially as the identity matrix, and so can be thought of partly as a correction of the network's output prior to that round (though the remainder of the network continues to evolve). The x-axis is the output nodes, and the y-axis is the nodes of the previous layer. Note for example that the last 26 columns look pretty different; these are the predictors for the 26 letters, which occur in the output after the SDF pixels. Green means positive and red means negative, so if you are looking at this in a black-and-white printout, that may explain your current confusion. The exact diagonal is a strong green, close to 1.0, although over time these weights do diverge from the identity somewhat. In the bottom-right corner of size 26^2, we are looking at how the 26 letter predictors are derived from the previous layer's predictions. We see that most letters are negatively correlated (makes sense; only one will ever be 1.0) although there are some oddities (probably because it found some other, better correlates). All nodes on this new layer have dense references to these 26 predictions on the previous; this means that the bottom 26 rows kind of represent biases for each of the 26 letters (what does an average 'e' look like?). I also included a dense region above that, but this appears to have simply evolved the same way as other 36^2 chunks have (the rest are sparse). These chunks have a large amount of spatial similarity (suggesting that the sparse sampling would be adequate), with a meaning like "if this area of the image is bright, then this pixel should be less bright." It is interesting that the pixels immediately next to the diagonal are almost always strongly negative (sharpening operation). "Thank you for attending my TED talk." —Figure 14

clockwise (simple case analysis on the three pixels ahead of me). As I walk the edge, I look at the normal (orthographic as we are doing orthography) of the edge and see where it reaches the edge value on the SDF; this point (a float) is output into the contour. The process is guaranteed to return to where we started. I recurse by negating the SDF (outside becomes inside) and bitmap (land becomes sea), and reverse the winding order of the result of recursion.

This gives me a perfectly fine line-based outline of the SDF's shape. Since I output points at every pixel, sometimes these points are inefficient (e.g. a series of colinear points on a straight line), and sometimes they reflect sharp corners that are not aesthetic. So I then take a second pass at each contour, and try fitting Bézier curves to sequences of points while the error remains low. Again I did fitting with a black-box optimizer, which is nice. However, the function being minimized also needs to be able to find the closest point on a Bézier curve to another point, and although this can also be done easily with the black-box optimizer, nesting an optimizer invocation inside another one proved to be way too slow. I found an old algorithm in a book I owned and was stymied by as a child [9].

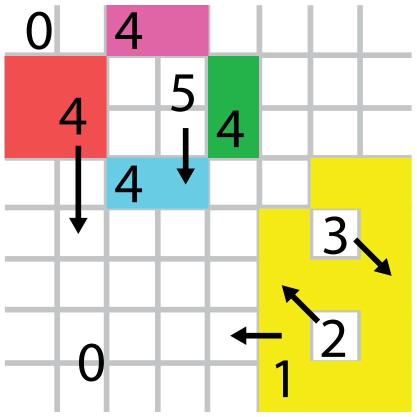


Figure 15: Some tricky cases to think about when generating the nested connected components, as the first step of tracing SDFs. Area **0** is the outside of the entire letterform, but note that we should include the top-left corner even though it is not reachable without leaving the bounds of the image. Area **1**'s parent is 0; it has two holes within it, Areas **2** and **3**. At the top left, the four pixel chunks making up area **4** are not actually connected, but they separate the hole which is child are **5**. This hole must have one parent, so it means that all four pixel chunks are part of the same area **4**.

Now we're all set up to take an input shape (e.g. from an existing font), run the make_uppercase or make_lowercase model(s) on it, maybe multiple times, and trace the resulting SDF into a vector form that can be used in a font. I did this on the canonical sans serif font Helvetica [14] and serif font Times New Roman [15]. Each font is produced by a symmetric process; for example, to make a font "more lowercase," I take the input font's lowercase alphabet and run make_lowercase on it (this becomes the output

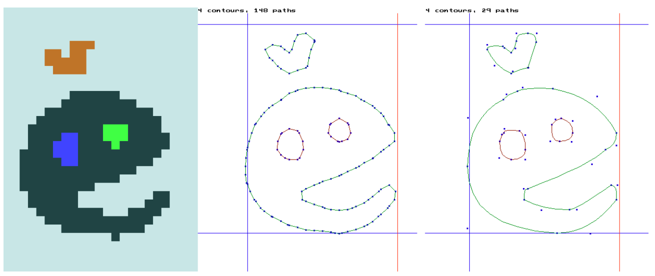


Figure 16: Tracing the SDF from Figure 10 into vector format. Left image shows the nested connected components. Middle image is the initial straight-line trace, and the right image shows the simplified contours using quadratic Béziers.

font's lowercase letters), and then run make_uppercase on *those* to produce the output font's uppercase letters. These letters are usually recognizable as the "normal" lowercase letters even though they've been through both neural networks. Before tracing, I do some automatic gamma adjustment of the SDFs (e.g. at least 5% of the pixels should be above the threshold), as the unadjusted letters seemed a bit too light. These fonts can also be downloaded from `http://tom7.org/lowercase/` for your corporate PowerPoint needs.

Helvetica means "of Hell", so making the font more uppercase give us **Heavenica** (Figure 17), since Heaven is "up" from Hell. What's lower than Hell? Spezial Hell,[6] as in "There's a Spezial Hell for the scalpers and cryptocurrency environmental terrorists stockpiling GeForce 3000 series GPUs so that I can *not* just get *one* darn card at a reasonable price for my important SIGBOVIK experiments." So the extra-lowercase version of Helvetica is **Spezial Helvetica** (also Figure 17).

Times New Roman refers to the multiplication operator in algebra, which has a natural uppercase in exponentiation. Thus the uppercase version of Times New Roman is **Exponential New Roman**. Computing Tetration New Roman or $\uparrow\uparrow\uparrow\uparrow$ New Roman [11] is straightforward, but extremely punctilious SIGBOVIK page limits preclude showing them here. Of course the lowercase version is **Plus New Roman** (and similarly implies Successor New Roman). Both fonts are shown in Figure 18.

The vector-based **Futurda** font (Figure 9) is in some ways more readable than these, but for the sake of comparison, note that these fonts are actually doing something more interesting, as they are built with both the make_uppercase and make_lowercase models. Futurda's $\boxed{\text{A}}$–$\boxed{\text{Z}}$ are just the lowercase of existing uppercase letters, which already has a correct solution and which a ML model can simply learn through memorization. In contrast, none of the letterforms in Heavenica can come through memorization of a training example (at worst, memorizing an

[6] I first learned about Spezial Hell from a Rugen Bräu beer that I drank in the Alps in Grindelwald, Switzerland (*la Confédération Hélvetique*).

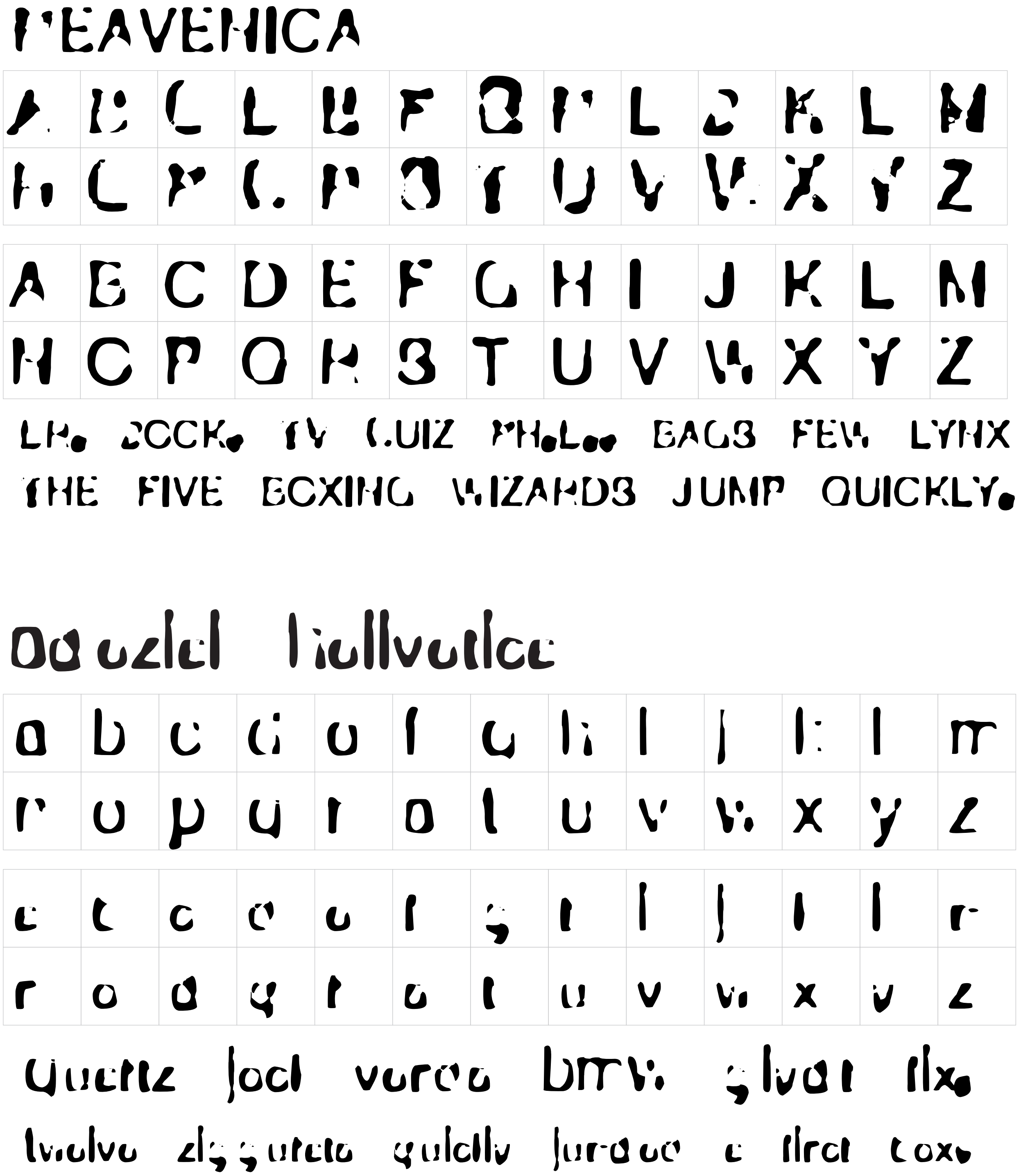


Figure 17: Type specimens for the generated font **Heavenica** (top) and **Spezial Hellvetica** (bottom). The uppercase letters in Heavenica are make_uppercase applied to uppercase letters from Helvetica, and the lowercase are make_lowercase applied to those. These lower-upperercase letters resemble regular uppercase letters, as they should; this gives you some idea of the quality of the model. Spezial Hellvetica is the symmetric thing (its lowercase letters are make_lowercase of Helvetica's lowercase). The sample text in this latter case is "Quartz jock vends BMW glyph fix. Twelve ziggurats quickly jumped a finch box."

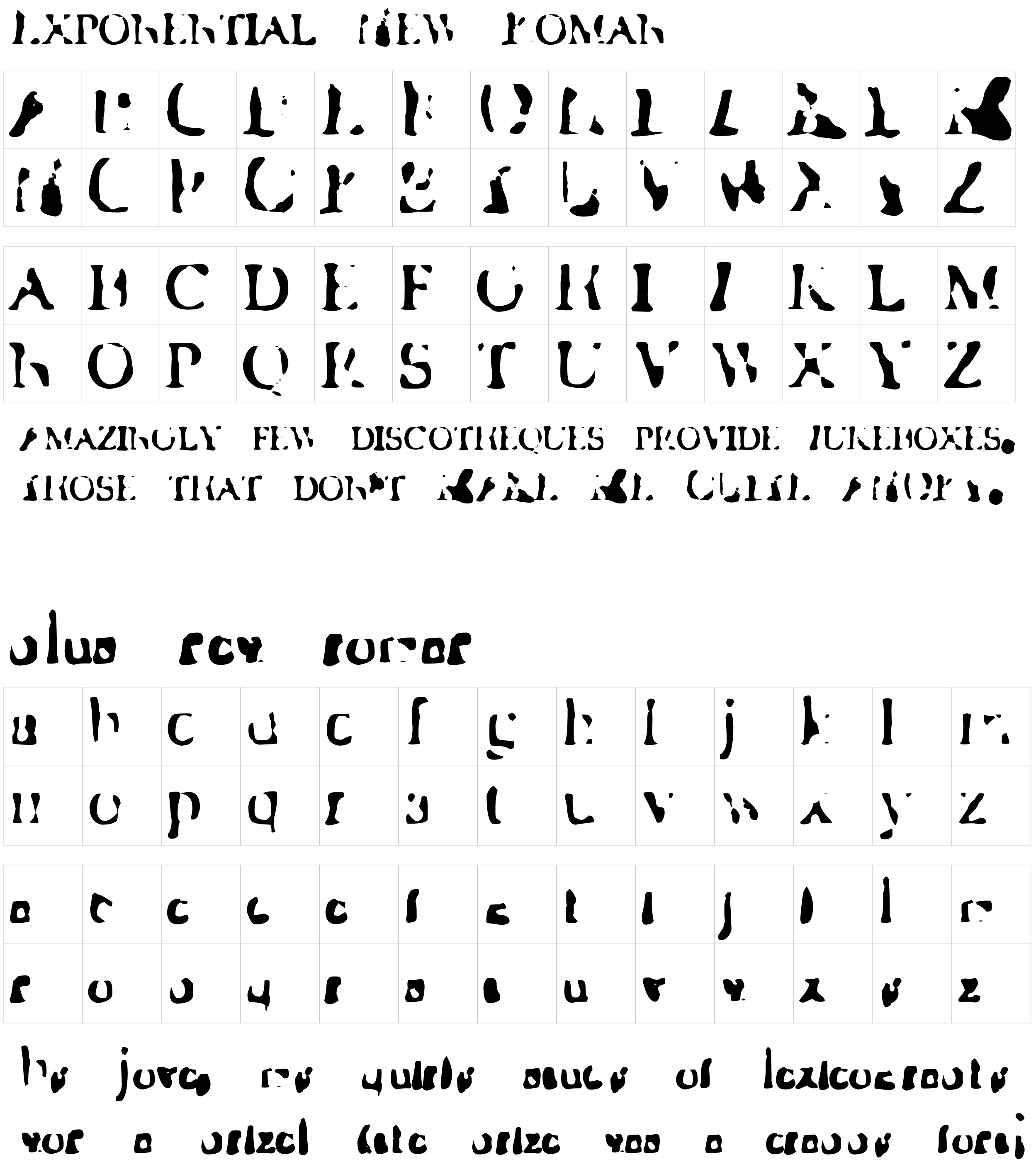


Figure 18: Type specimens for the generated font **Exponential New Roman** (top) and **Plus New Roman** (bottom). These were produced with the same procedure as in Figure 17, but starting with Times New Roman. The letterforms are clearly different, so it's not as though the models are (just) memorizing a shape for each letter. Notably, smudgy serifs reappear when the uppererercase letters are re-lowercased, as desired. Sample text here is "Amazingly few discotheques provide jukeboxes. Those that don't MAKE ME QUITE ANGRY." and "By Jove, my quirky study of lexicography won a prize! (the prize was a crappy font)"

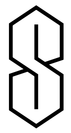
inversion example generated by the other model after *it* memorized something). Subjectively, the fonts are not very readable and only slightly interesting, but the two models did demonstrate a reasonable ability to invert one another's behavior.

7 Perfect letters, hallucinated

Oh, you don't like letters that look *bad*? Instead you want letters that look *good*? How about *best*?

The fonts in the previous section were created by modifying the case of existing letterforms, with mixed success. We can also do this to any letter-like shape. I built a UI for drawing letters and seeing them uppercased and lowercased (and then re-lowercased and re-uppercased) live, but it's impossible to demonstrate in paper form. It's pretty much what you'd expect.

The UI also tells you how much your input resembles the various existing letters $\boxed{A}$–$\boxed{Z}$ and $\boxed{a}$–$\boxed{z}$ using the 26 bonus outputs that each model predicts. For example, I learned that the "Cool S" [20]:



does not much resemble an S.

This begs us to ask the question: What shapes *do* look like letters? Since the models will tell us, I can just search over shapes and ask them. The first thing I tried was to just generate random shapes and optimize their parameters to produce the highest possible prediction for the target letter, and the lowest possible prediction for the rest. This produced results that are fully bonkers (Figure 19).

We can improve the results by searching for inputs with a "perfect" prediction (1.0) rather than making it as high as possible. These results may not have been fully bonkers, but were at least downright wacky. Since there appear to be a large variety of inputs that the model judges as "perfect", the most appealing results from this excursion involved scoring some additional properties of the hallucinated inputs to discourage them from being so barmy. I generated 8×8 bitmaps, plenty of pixels to make readable letters on classic computers. Rather than allowing them to be arbitrarily noisy, I also weakly optimized for (1) the number of pixels set being close to half and (2) minimal transitions between on and off along each row and column. This produced shapes that are basically letter-like, but weird (Figure 20).

8 Chess-playing

One obvious thing to do with a program that takes an 8×8 bitmap and produces some kind of score for it is to use that program to play chess [17]. Here I entered 26 such programs in the Elo World tournament [18], which allows us to see how they perform against each other and benchmark algorithms. (Badly.)

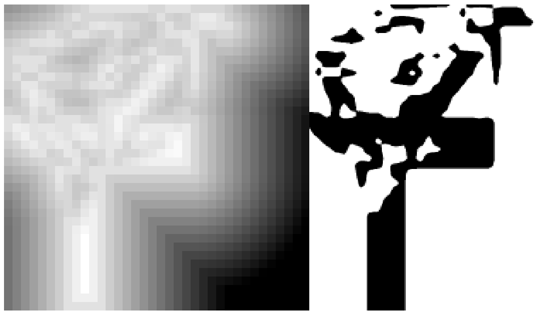


Figure 19: A randomly generated SDF (left) and its rasterization (right) which maximized the predicted score for $\boxed{F}$ (1.893 out of a nominal 1.0) while scoring all other letters low. Parts are recognizable as an $\boxed{F}$, but other parts are fully bonkers. This is actually one of the least weird ones.

At each turn, the algorithm takes the board state that would result from each legal move, and interprets it as an 8×8 bitmap. It renders that bitmap as an SDF and then runs the make_lowercase model on it, and chooses the move that minimizes the difference between its letter predictions and the $[0, 0, \ldots 1, \ldots 0]$ vector selecting the letter we are "playing as." det asym

After tens of thousands of games each, it is clear that the letter-based players are all bad at chess. Letters $\boxed{E}$ and $\boxed{S}$ perform the worst (agreed on $\boxed{S}$ being the worst, thank you very much!), even worse than the "No, I insist!" strategy that tries to force its opponent to capture its pieces. The letter $\boxed{T}$ (eponymous! yeah!) performs best, but still worse than random. The numeric players like π and e are categorically better than the alphabetical ones, but this is not surprising because chess is more of a mathematical game than a linguistic one (Figure 21).

The abbreviated tournament results:

name	**elo**	**wins**	**losses**	**draws**
worstfish	395.95	10	44406	18584
...				
letter e	602.27	1717	22566	38717
letter s	602.92	1310	22020	39670
no i insist	605.47	0	20405	42595
letter f	605.60	1341	21653	40006
letter y	606.00	1347	21688	39965
letter b	607.39	1705	21870	39425
letter p	607.61	1790	21963	39247
letter l	608.75	1370	21306	40324
letter j	610.86	1737	21525	39738
letter u	611.41	1363	20947	40690
letter c	612.38	1264	20787	40949
huddle	612.60	1172	20494	41334
letter n	612.81	1395	20802	40803
letter w	613.64	1342	20723	40935
letter g	613.72	1339	20660	41001
letter v	614.18	1390	20606	41004
letter h	615.04	1328	20452	41220
letter r	615.22	1826	20932	40242
letter x	615.70	1414	20457	41129
letter m	616.31	1774	20809	40417
letter q	618.85	1378	19993	41629
letter o	619.08	1502	20078	41420
letter z	620.09	1745	20275	40980
letter d	620.10	1730	20369	40901
...				

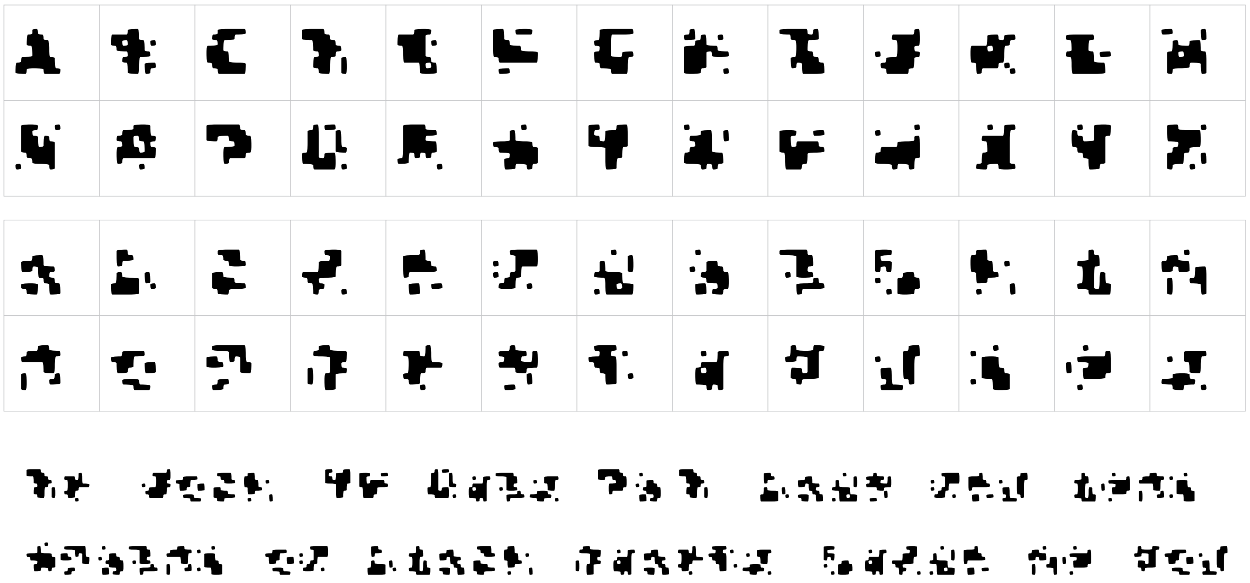


Figure 20: Type specimen for the generated font **Perfect Hallucination**. Each letter is an 8x8 bitmap that looks as close to "perfect" as possible to the model. Perfect here means that for $\boxed{C}$, the make_lowercase model outputs as close to the vector $[0, 0, 1, 0, 0, \ldots, 0]$ (in its 26 letter predictors; the actual lowercasing is ignored) as the optimizer could find. (Of course I didn't search all 2^{64} inputs, but errors are on the order of one part per thousand). The models are completely successful at recognizing normal-looking letters as well, but it likes these even better.

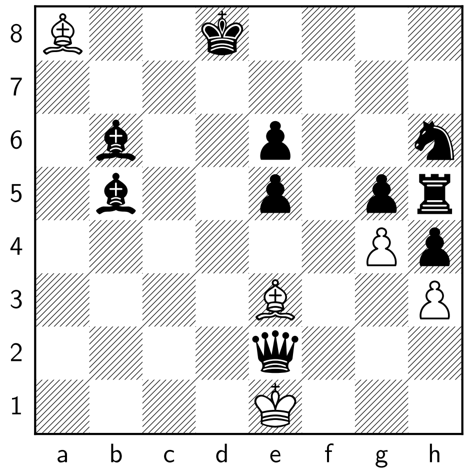


Figure 21: The letter $\boxed{P}$ (white) loses to the numeric constant π (black) after 59 nonsensical moves. The $\boxed{P}$ player tries to move pieces such that the board looks like a letter $\boxed{P}$ (and no other) to the neural network. The π player uses $3 - \pi$ to arithmetically decode the sequence of legal moves (sorted alphabetically by PGN). Neither player is concerned with chess, really, but the letter-based players are generally bad because they are more likely to get stuck in local minima once they are basically happy with the shape of the pieces.

...	elo	wins	losses	draws
letter a	620.57	1819	20212	40969
letter i	622.50	2169	20424	40407
letter k	622.90	1755	19878	41367
letter t	623.30	2237	20271	40492
...				
random move	655.90	7267	20983	34750
...				
chessmaster.nes lv1	1014.74	37302	13992	11706
...				
stockfish1m	2831.66	60167	493	2340

9 Other Applications

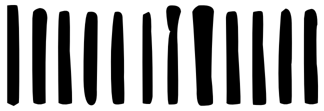
People often stop me on the street to ask, Tom, Why did you spend so much time and energy on this useless SIGBOVIK project? To which I say, Ha! At least I am not wasting my time *reading* SIGBOVIK papers or talking to strangers on the street! and run off. Although the main purpose of SIGBOVIK is to confound bibliometrics with ambiguously good-faith and high-quality research published in a clearly satirical but superficially decorous venue associated with a traditionally esteemed university, it is also possible for such work to have practical applications in arts and spycraft. You simply need to give it some thought.

For example, it is well known to internet trolls and DOMAIN NAME PHISHERS that the uppercase $\boxed{i}$ and lowercase $\boxed{L}$ are indistinguishable in many fonts, allowing for various Tomfoolery.[7] This can also be used for steganography—hiding messages inside text without the use of em dashes—by selectively replacing letters with their

[7] I actually wrote "troiis" and "domaln name phlshers", hehe!

alternates. Each replacement only encodes one bit of information, however. With the generic ability to uppercase and lowercase letterforms, we can exploit this ambiguity to generate a large variety of letterforms that can be used like this.

For example, the following sequence of distinct letters are hard to distinguish from one another and could all be used in place of a lowercase $\boxed{\mathrm{L}}$ or uppercase $\boxed{\mathrm{i}}$:



The letterforms are generated by repeated application of the make_uppercase (↑) and make_lowercase (↓) networks to the lowercase $\boxed{\mathrm{l}}$ from Helvetica. They are, from left to right: $\boxed{\mathrm{l}}$, ↑↓ $\boxed{\mathrm{l}}$, ↓↑↑↓ $\boxed{\mathrm{l}}$, ↑↓↑↓ $\boxed{\mathrm{l}}$, ↑↑↓↓ $\boxed{\mathrm{l}}$, ↓↓↑↑↓ $\boxed{\mathrm{l}}$, ↓↑↑↓↓ $\boxed{\mathrm{l}}$, ↑↑↑↓↓ $\boxed{\mathrm{l}}$, ↓↑↓↑↑↓ $\boxed{\mathrm{l}}$, ↓↑↑↓↑↓ $\boxed{\mathrm{l}}$, ↓↑↑↑↓↓ $\boxed{\mathrm{l}}$, ↑↓↓↑↑↓ $\boxed{\mathrm{l}}$.

In this way we can encode much longer bit strings in a single character, even thousands of iterations deep if there is a reasonable balance of uppercasing and lowercasing operations. (Too many in a row will get us stuck; see Section 10.) Not all sequences lead to a shape like this (commonly they resemble $\boxed{\mathrm{i}}$ or $\boxed{\mathrm{L}}$ when starting from $\boxed{\mathrm{l}}$), but we can easily create a codebook of ones that do. Maybe I have even hidden an intricate message in this paper for you to discover? (I didn't.)

10 To infinity, but let's stop there

Wow, this project is pretty involved, huh? Let's just add another dimension to it!

We looked at what we get when we run make_lowercase on an existing lowercase letter, making it lowerercase. Of course, we can run the model again, and get an even lowerercase letter. The process can be repeated indefinitely. As it turns out, lowercasing tends to make letters smaller and smaller (makes sense) and they eventually just turn to dust (Figure 22) and stay that way (makes sense; "dust to dust" [5]).

It's possible to make the results a bit more interesting by injecting additional energy after each iteration, either by adjusting the gamma or "zooming" into the active area. This seems pretty arbitrary, though. I think it is right to conclude that the lowestcase versions of letters are canonically specks of dust.

On the other hand, repeatedly uppercasing produces more interesting results. Uppercasing usually increases the scale of a letter, but this effect is limited by the finiteness of the SDF and the fact that the outer edges are very unlikely to have high values. Moreover, although uppercase letters are large, they also have a lot of internal space. So iterations do not simply grow in size or fill the space, but repeatedly grow and deteriorate like an organism in the Game of Life [8]. Animating the SDF under iterations of

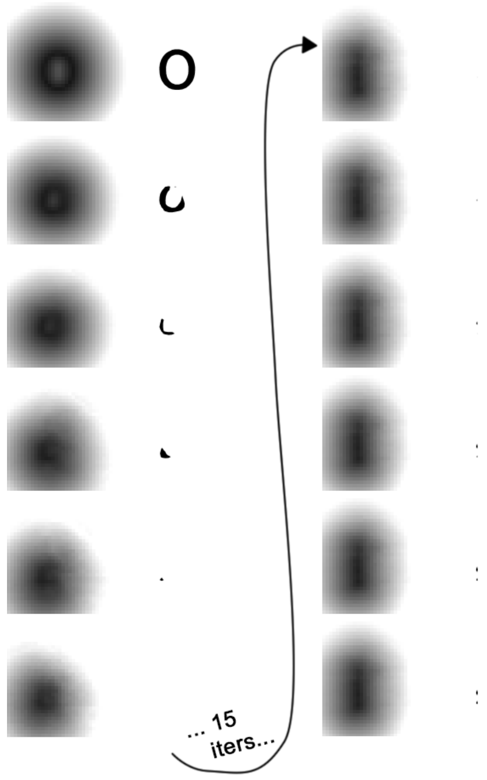


Figure 22: The first 27 iterations of the make_lowercase model on the letter $\boxed{\mathrm{o}}$. The lowercase model generally makes letters get smaller and smaller until they disappear. There is still energy in the SDF (left column) but no pixels exceed the threshold so the rasterization is empty (right column) for about 16 iterations. Finally it reaches a stable state, a tiny piece of colon-shaped dust, easily mistaken for a printing error. All letters (from the test font Helvetica) converge to this shape, except mysteriously the letter $\boxed{\mathrm{v}}$. Is $\boxed{\mathrm{v}}$ a different alien species, masquerading as a normal letter for millions of years?

the model is reminiscent of a flickering candle, recalling another well-known approach to increasing the emphasis of text, `flamingtext.com`. Is this eternal flame itself the uppestcase letter? Alas, such an effect is not possible in print (paper is too flammable).

Iterating the make_uppercase model with 32-bit floats does not form any cycles in 25,000,000 iterations, nor does it if the intermediates are quantized to 8-bit ints. This is curious because under visual observation the sequence does appear periodic, and in fact seems to be the same characteristic loop reached by all letters. Presumably there are some oscillations with long, relatively prime periods or even some pixels that are monotonically growing or shrinking, but very slowly. However, if I render the SDF as a 1-bit bitmap at 2× scale, I get a perfect cycle of 132 frames. This appears to be two passes through the main characteristic loop, but slightly different each time. This is not a strange property for a letter to have; for example a typical $\boxed{\mathrm{B}}$ has two copies of the same basic idea in it.

None of these alone could be considered the uppestcase letter, but perhaps together they are? A natural way to include them all in one shape is to stack them in 3D, as if each iteration is an MRI slice of the brachial plexus.

To generate a 3D mesh, I stack the 132 SDFs in the z direction, and this naturally yields a three-dimensional signed distance field (trilinear interpolation). The classic "marching cubes" algorithm [13] for mesh generation works natively on such a field, and come to think of it, I probably could have used that in two dimensions to trace the SDFs to vectors. Oh, well. Fortunately there is enough spatial similarity between the slices that it makes a reasonable 3D shape even without interpolation, but sampled at a decent resolution and then cleaned up, it looks quite nice; much better than the lowercase colon. Speaking of colons, it rates approximately a 2.5 on the Bristol stool scale [12]. A 2D projection of the manifold is in Figure 23. I 3D-printed it and am currently working on a way of embedding it as an unusually tall key on my keyboard that I can press whenever I wanted to express ULTIMATE ANGER.

11 Conclusion

We performed an exhaustive case analysis, exploring cases both more upper- and lower- than ever been seen before. Sideways case was not considered, as that is nonsense. We had modest success generating uppperercase and lowerercase letterforms through two neural network models simultaneously trained to be each other's inverse, although frankly it was much faster and more aesthetic to just do it by hand. After really following through on the downloadables and taking some needless excursions for effect, we saw that these models have limits (at least informally). The lowestcase letter is already on your keyboard; it is the ASCII eyeballs character $\boxed{:}$. The uppestcase letter is not so easily typed, and perhaps that is for the best.

Futura Work. I think I pretty much beat this one to death, honestly.

Acknowledgements. For this project I used `stb_truetype.h` from the excellent stb collection of single-file libraries [3]. I did push its limits somewhat and "automatically discovered" assertion failures and other crashes (e.g. during blackbox optimization on all 100k fonts), but it saved a lot of time. This library only helps with reading the fonts and generating SDFs. To generate TTFs, I had to write my own pipeline, which generates FontForge's .SFD (a typo factory right there) files, and then did the final export with FontForge [6]. Thanks Jason Reed for this suggestion, without which I would probably still be developing my own TTF file writer and custom hinting engine. Finally, I would like to thank the pseudonymous SIGBOVIK program committee for overseeing the proceedings, and the SIGBOVIK program committee committee for overseeing *them.*

References

[1] Tom 7. Divide by Zero fonts, 1993. `http://fonts.tom7.com/`.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques.* Addison Wesley, 1986.

[3] Sean Barrett. stb single-file libraries, 2009–2020. `https://github.com/nothings/stb`.

[4] Vincent Connare. Comic Sans, 1994. `https://en.wikipedia.org/wiki/Comic_Sans`.

[5] Thomas Cranmer. Burial of the Dead, Rite Two: The Committal. In *The Book of Common Prayer.* The Archbishop of Canterbury, 1552.

[6] George Williams et al. Fontforge, 2000–2015. `https://fontforge.org/`.

[7] Person Famous, and presumably rich. A neural network paper that everyone cites, Beforetimes. Probably AAAI or NIPS, idk.

[8] Martin Gardner. The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223:120–123, 1970.

[9] Andrew S. Glassner. *Graphics Gems.* Acadmic Press, Cambridge, MA, 1990.

[10] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *Advanced Real-Time Rendering in 3D Graphics and Games, ACM SIGGRAPH 2007 Course 28*, pages 9–18. ACM, 2007.

[11] Donald E. Knuth. Mathematics and computer science: Coping with finiteness. *Science*, 194(4271):1235–1242, December 1976.

[12] S. J. Lewis and K. W. Heaton. Stool form scale as a useful guide to intestinal transit time. *Scandinavian Journal of Gastroenterology*, 32(9):920–924, 1997.

[13] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, August 1987.

[14] Max Miedinger. Helvetica, 1957. `https://en.wikipedia.org/wiki/Helvetica`.

[15] Stanley Morison. Times New Roman, 1931. `https://en.wikipedia.org/wiki/Times_New_Roman`.

[16] Tom Murphy, VII. Red i removal with artificial retina networks. In *A record of the proceedings of SIGBOVIK 2015*, pages 27–32. ACH, April 2015. `http://sigbovik.org/2015`.

[17] Tom Murphy, VII. Color- and piece-blind chess. In *A Record of the Proceedings of SIGBOVIK 2019.* ACH, April 2019. `http://sigbovik.org/2019`.

[18] Tom Murphy, VII. Elo World: A framework for benchmarking weak chess algorithms. In *A Record of the Proceedings of SIGBOVIK 2019*. ACH, April 2019. `http://sigbovik.org/2019`.

[19] Paul Renner. Futura, 1927. `https://en.wikipedia.org/wiki/Futura_(typeface)`.

[20] Unknown. Cool S. `https://en.wikipedia.org/wiki/Cool_S`.

[21] Akelsey Vaneev. BITEOPT – derivative-free optimization method, 2021. `https://github.com/avaneev/biteopt`.

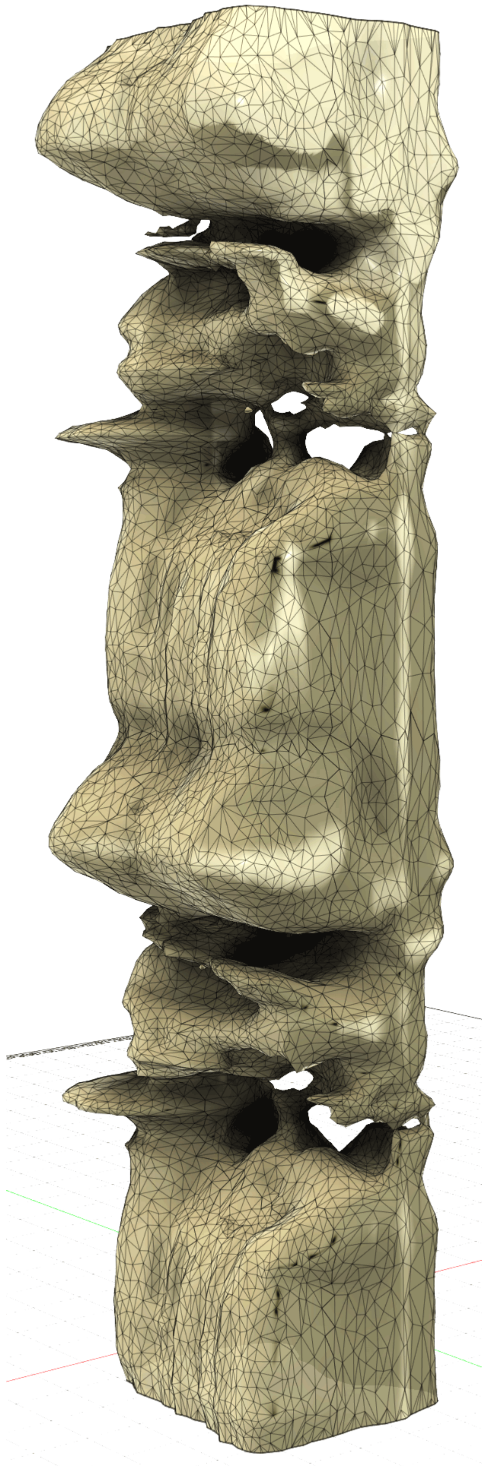


Figure 23: 3D manifold showing a section of the repeating loop as the make_uppercase model iteratively uppercases a letter. (Shown here is the input $\boxed{q}$ from iteration 245–377, but they all converge to this same periodic shape.) Slices through this shape give a letterform's outline (or usually, a linear interpolation between two of them). The bottom of the shape is its "beginning" but it appears to repeat like this forever.