Winning Chess Hackathons For Fun and Profit!

George Williams · Follow
13 min read · Jan 20, 2025

Part I: The Data and The Model

Introduction

I recently participated in Strong Compute's last AI Chess Hackathon in December 2024. Honestly I really had no business getting involved. My last hackathon was years ago and, perhaps more importantly, I'm not a great chess player. That said,

machine learning has become very powerful lately and I was curious if I could train a deep learning model based on convolutional neural networks or transformers that could play chess well. I've leveraged machine learning for all sorts of challenging problem domains — computer vision, natural language, drug discovery, bioinformatics, radio signal processing, sports analytics, motion capture, robotics- but never for games such as chess.

As it turns out, I ended up winning a trophy— beating all other newbie competitors in the final tournament. In addition, I also qualified to get a bunch of free credits to use Strong's massive GPU cluster!

How did I do it you might be asking? Well that's what this 3-part blog series is about. These blogs are targeted to data scientists, AI engineers, even software engineers who have a basic understand of machine learning (preferably who have some experience running and training models.) Some experience with chess will be useful, and, of course, you should have an interest in training a chess playing AI. Who knows, maybe my notes here could help you win a chess hackathon too!

Here is the outline of the series:

- In *Part 1: The Data and The Model* I'll start with some basic background. We'll get into the PGN chess data format, chess scoring, relevant deep learning primitives, and the model architecture that I trained.
- In *Part 2: Training and Inference* I'll get into the details of training the model in PyTorch and I'll show the game results of the trained model playing against itself. This will give me a chance to discuss how chess players (either human or AI) are rated and ranked.
- In *Part 3: Scaling Chess AI* I'll dive into some of the latest bleeding chess AI research, including DeepMind's release of the most comprehensive chess training set ever curated. I'll show you how you can train state-of-the-art generative AI models leveraging Strong Compute's easy-to-use distributed GPU tools and infrastructure.

This will culminate in a final virtual tournament in which I'll have all the models developed in this blog series compete against each other.

A Very Brief History Of Chess And Computing

IBM's Deep Blue in the Computer History Museum in Mountain View, CA.

Before we get into it, let's take a step back. Since the beginning of computing in the 1940s, the development of competitive chess engines has been a singular goal for software/hardware engineers and computer scientists. Chess has also served as the backdrop for key milestones in computing, including:

- *1948* Alan Turing developed the first algorithm and program for playing chess. Unfortunately no hardware powerful enough to run it was available until after Turing died in 1954! Turing developed the first methods to "score" potential chess moves (e.g., determine the best move to make given current state of the board.)

- *1985* Carnegie Melon University began work on ChipTest, the first chess engine designed from the ground up including custom silicon, micro-code, and a chess tree-based search algorithm that could score 50K moves per second.

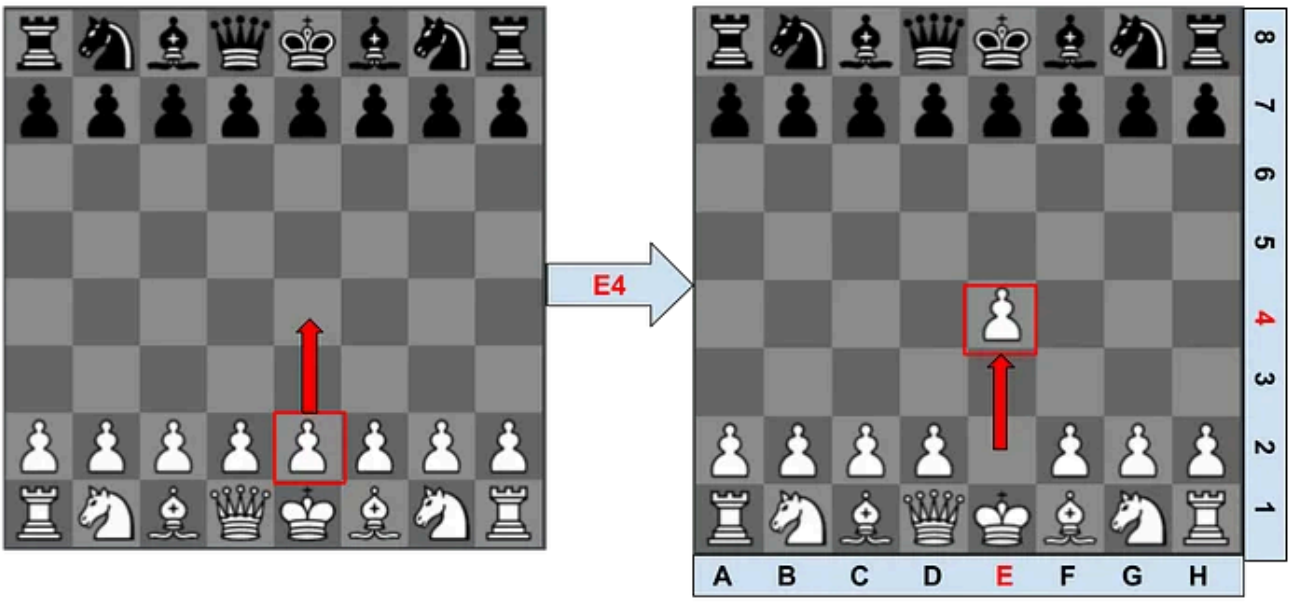
- *1997* IBM's Deep Blue beat Gary Kasparov becoming the first chess engine to beat the current (human) chess champion in a full match.
- *2017* DeepMind released AlphaZero which leveraged neural networks in addition to search trees. All top chess engines now use neural networks and deep learning.

Chess Game Data

OK, let's start with the data. If you thought that leveraging previously played chess games would be useful to train a chess playing AI, you are on the right track. Fortunately, there's a ton of openly available chess game play data out there.

One popular data format for chess games is called PGN, which stands for Portable Game Notation. It's a compact way to track chess moves and board states for an entire chess game from beginning to end.

The following is an example of the PGN for an opening chess move:



In this board, "E4" indicates one of white's pawns has moved forward two positions.

The move shown is encoded as the string "E4" which indicates the destination of a piece's move. In this case, a white pawn has moved forward two positions in the board. The encoding uses the 2D coordinate system of a chess board (board on the right.) One axis is sequential letters and the other is sequential numbers. Simple, right?

The format also accommodates some unique scenarios:

- A chess move that overtakes an opponent's piece is indicated with a preceding "x."

- If there is ambiguity because multiple pieces could be moved to the same spot on the board — this is indicated with a preceding letter that indicates the piece that moved, for example "Q" for the queen.

A PGN string is a sequence of all the individual move encodings of a game. Check out the following site to acquire a visually intuitive and interactive understanding of a complete game using PGN notation.

If you are wondering how one might represent the state of the board at any point in the game sequence, all you need to do is replay the PGN game string from the beginning to any point in the game you want. We'll need this because the model I trained takes a board state as input, and not a sequence of moves.

Scoring Chess Boards

In this hackathon, participants had to train a model that takes as input an arbitrary chess board state, returning a "score" as output. In the hackathon's final tournament, whenever it was the model's turn to move in a game — the hackathon's game engine would send to the model all potential board states representing all currently available legal moves. The game engine picked the move with the highest score.

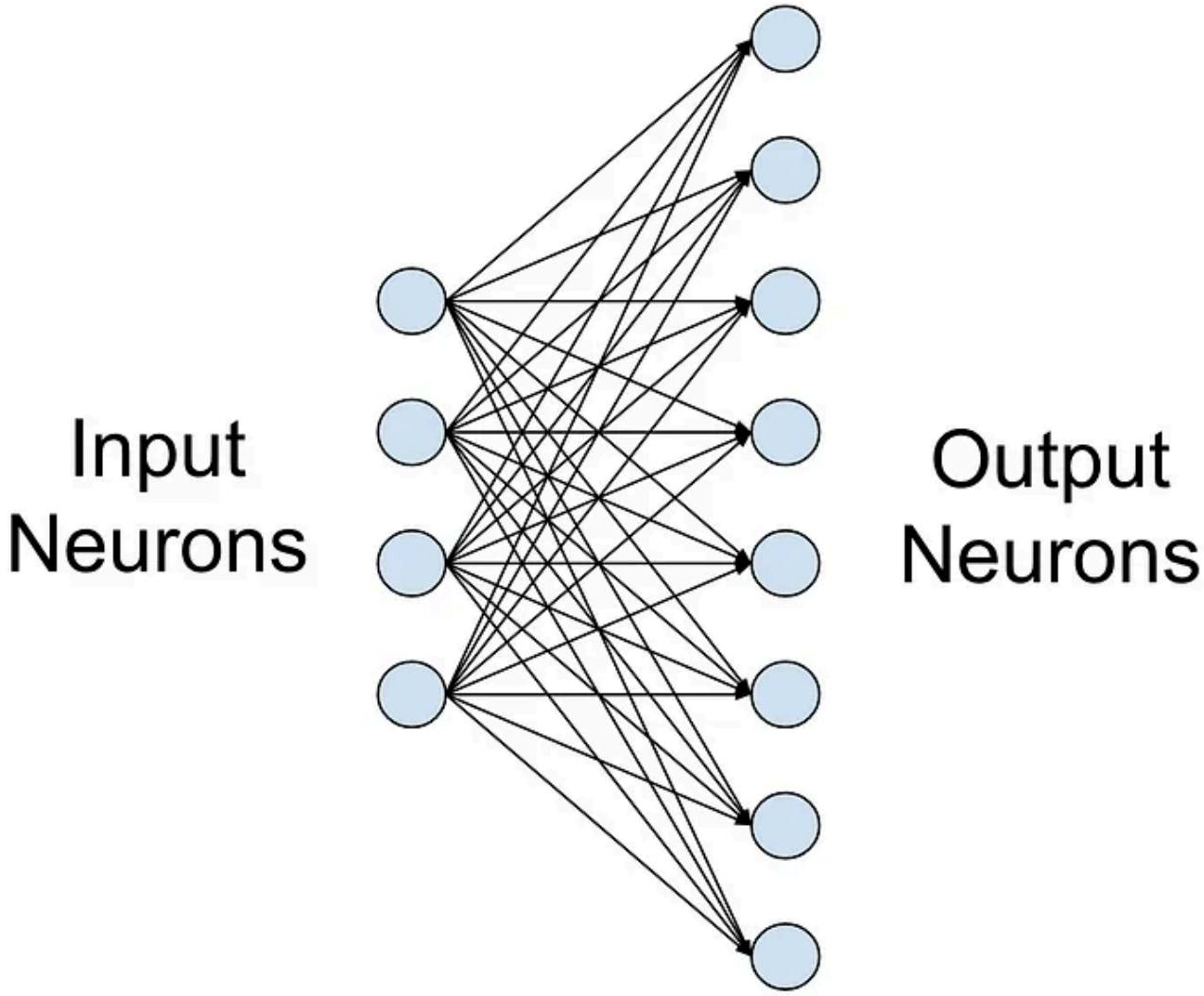
You may be wondering how ground truth scores were acquired for the training data. Fortunately, the hackathon provided various datasets for training, each with lots of pre-scored board states. I ended up using the dataset with scoring from one of the Leela Chess Zero chess playing engines. Leela Chess Zeros is a popular open source chess engine project and they train many of the best chess playing models that are openly available.

Deep Learning Primitives for Chess AI

Before getting into the model details, it's worth taking a moment to review the common deep learning primitives that compose most modern day neural networks. The following sections summarize the primitives I ended up using for my model exploration. Likely you are familiar with many of these concepts and I won't go into a ton of detail. Rather, I'll highlight concepts that are potentially useful when building a neural network that can play chess.

Linear Layer

Every neural network has a linear layer somewhere in it, so let's start here. The image below demonstrates a small linear layer. In this example, the layer takes a vector of 4 numbers on the left and outputs a vector of 8 numbers on the right. Notice that each input "neuron" connects to all output neurons via an edge connection. Each edge represents a constant "parameter" or "weight."



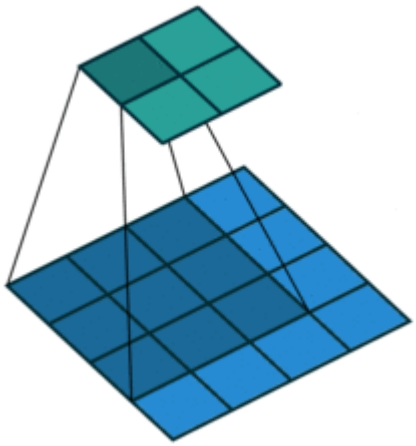
A simple Linear Layer which transforms a vector of 4 elements into a new vector of 8 elements.

The value of each output neuron is the sum of all input neurons connected to it, where each connected input neuron is multiplied by the edge's weight. Another way to think of the transformation from the input vector to the output vector as a matrix multiplication of the input vector by the implicit weight matrix.

These days, it's not unusual for linear layers to be composed of hundreds or even thousands of both neurons and weights. That's a lot of multiplications! In LLMs. for example, most of the weights (and compute workload) lie in the linear layers.

Convolutional Layer

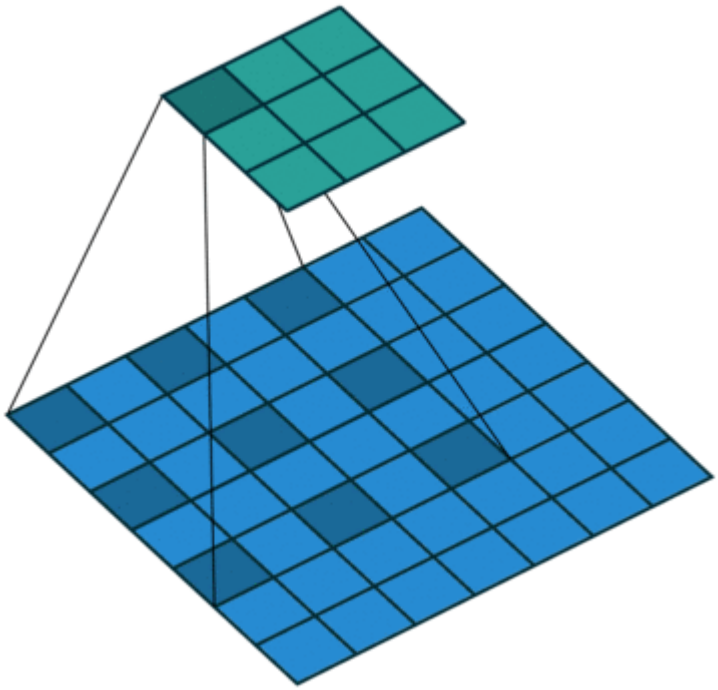
Computer vision deals with multi-dimensional data instead of just 1D vectors. For example, you can think of a black-and-white image as 2D matrix of gray scale data. A convolutional operation is depicted in the animation below. The convolution is a "sliding window" operation applied across the input (the 2D matrix shown in blue.)



A 3×3 convolution operation.

Notice here the final output of the completed operation is also 2D (in green.) The convolutional operator itself (depicted by the frustum) is actually just a small linear layer, in this case it has 3X3 input neurons and 1 output neuron.

There are many variants of the convolutional operator including 1D convolutions, 2D convolutions on multi-channel data (ie, consider a color image, each pixel composed of 3 elements), and higher dimensions convolutions like 3D and 4D. There are slightly different ways a convolution operation can traverse its input, for example by skipping pixels to reduce the computation requirements, or even skipping pixels of the input in each convolution operation as shown below (this is called "dilation.")

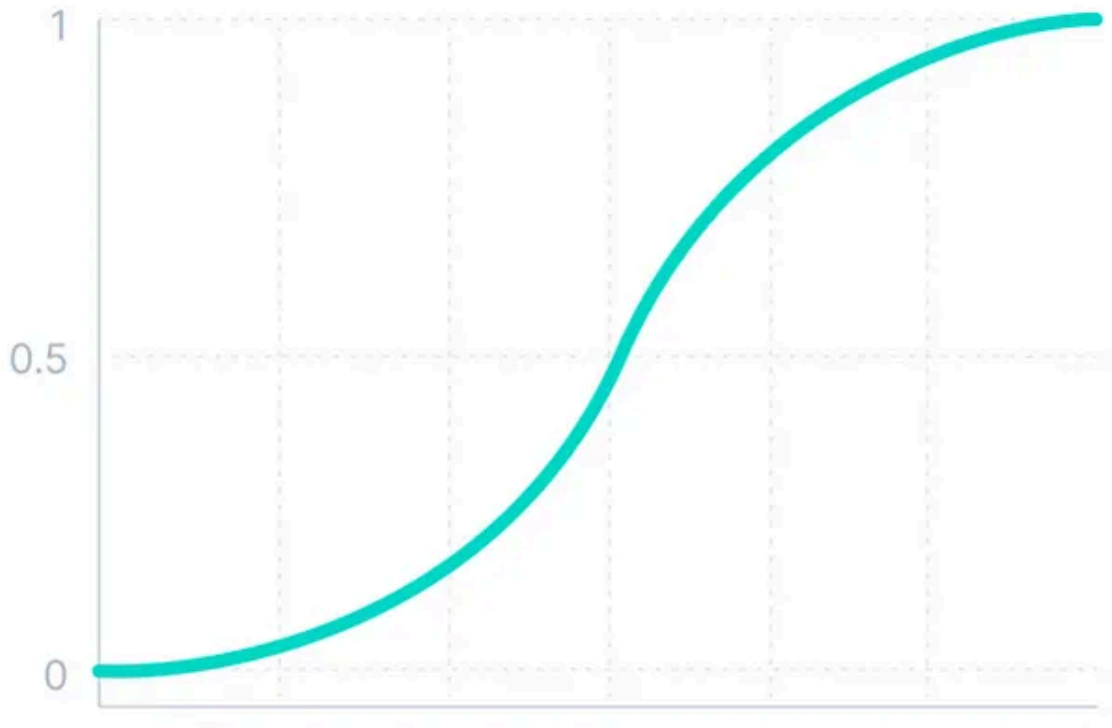


A dilated convolution skips pixels in each convolution.

A chess board is like an 8X8 image so its natural to include a convolutional layer within a chess playing neural network. The model I'll describe later takes advantage of several convolutional layers.

Activation Functions

At this point, you've probably noticed that outputs of linear and convolutional layers are the result of lots of multiplications. As inputs "feed forward" through the neural network, there are more layers and therefore more successive multiplications. This means there is the danger of numeric overflow. Activation functions can mitigate this, as well other types of numerical instabilities, by restricting neuron values to a specific range. When present, these functions are applied to each individual output neurons of a layer. Consider the sigmoidal activation function below, a very common activation function.



A sigmoidal activate function.

The range of possible values into this activation function is the full range of the x-axis but the function “squashes” the input to a range of values between 0 and 1 in a non-linear fashion. There are lot’s of different variants of activation functions and many of them perform this squashing. In addition to squashing and numeric stability, activation functions help the early stage of pre-training. The model I trained uses RELU (rectified linear unit) which is a simple function that passes positive values as-is, but clamps negative values to zero.

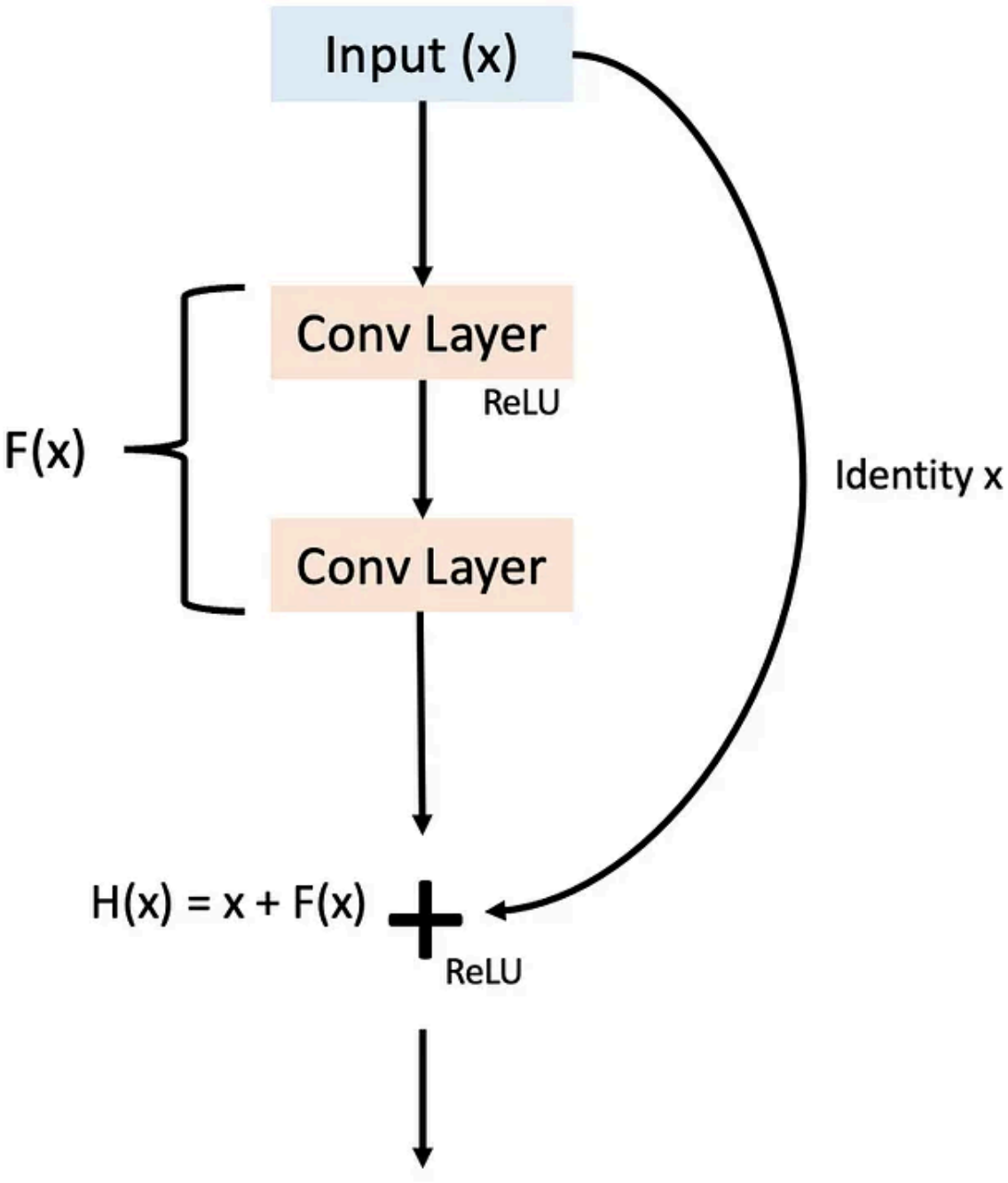
Normalization Layer

As you saw earlier, activation functions can have a normalizing effect on the data, but they don’t do this by capturing the statistics of the data itself. Batch normalization is a popular technique which performs this operation by looking at the range of data values for a subset of the training data at each step during training. And as opposed to activation functions, normalization layers can also have trainable parameters.

Residual Connection

A residual connection merely ties an input layer in one area of a neural network to an output layer somewhere else in the network. See the example below which shows the input layer “residually” connected to a layer further on down the network. What

does this do? Mathematically it's (typically) just a vector add operation — the input vector is just added element-wise to the destination's vector. In the example there are no parameters associated with the connection at all (that's what the "identity x" is emphasizing in the graphic.)



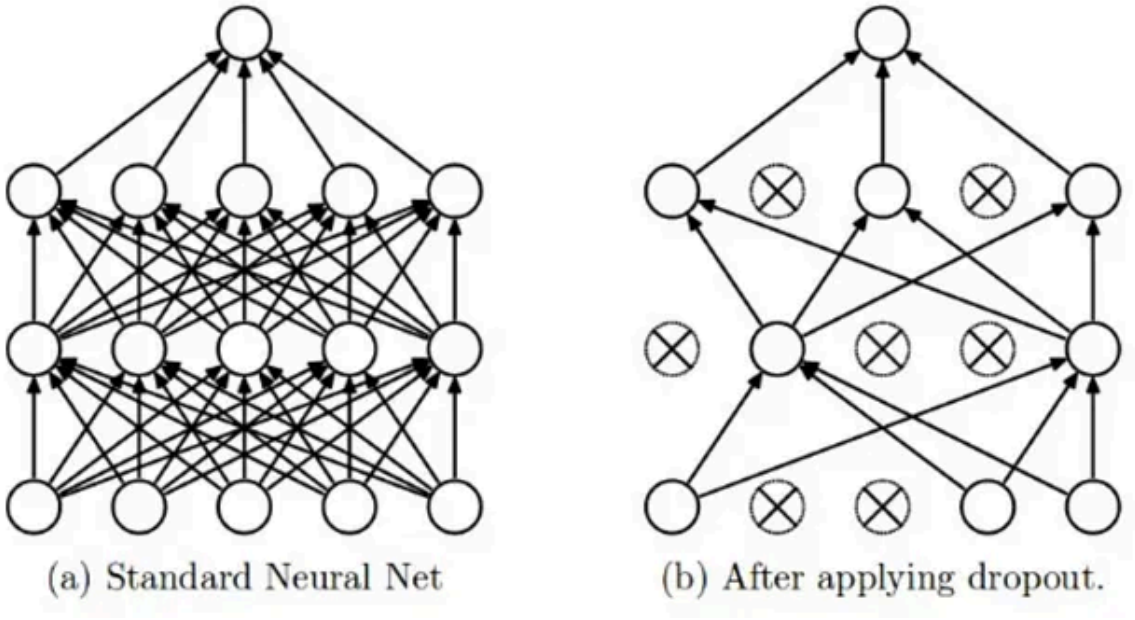
A residual connection from the input to the output of the second convolution layer.

Now, it doesn't seem like a useful primitive, but it is in fact one of the most important primitives of deep learning. Residual connections single- handedly enabled neural network designers to go "deeper" and craft neural networks composed of hundreds of layers. If you've ever used the model called ResNet, the

first convolutional neural network that leveraged residual connections, then you've used the deep learning model that ushered in the "Cambrian explosion" of deep learning in the early 2010s.

Drop-out Layer

You probably know that machine learning can "overfit" easily on the training data. An "overfit" model has memorized its training data to some degree. In other words, when this trained model is deployed, its predictions are only confident on model inputs that resemble closely the training samples. In order to help a model "generalize" instead of memorize, different techniques are available. One of those is to force some fraction of the model's weights to be zero randomly during training. And that's the role of a drop-out layer.

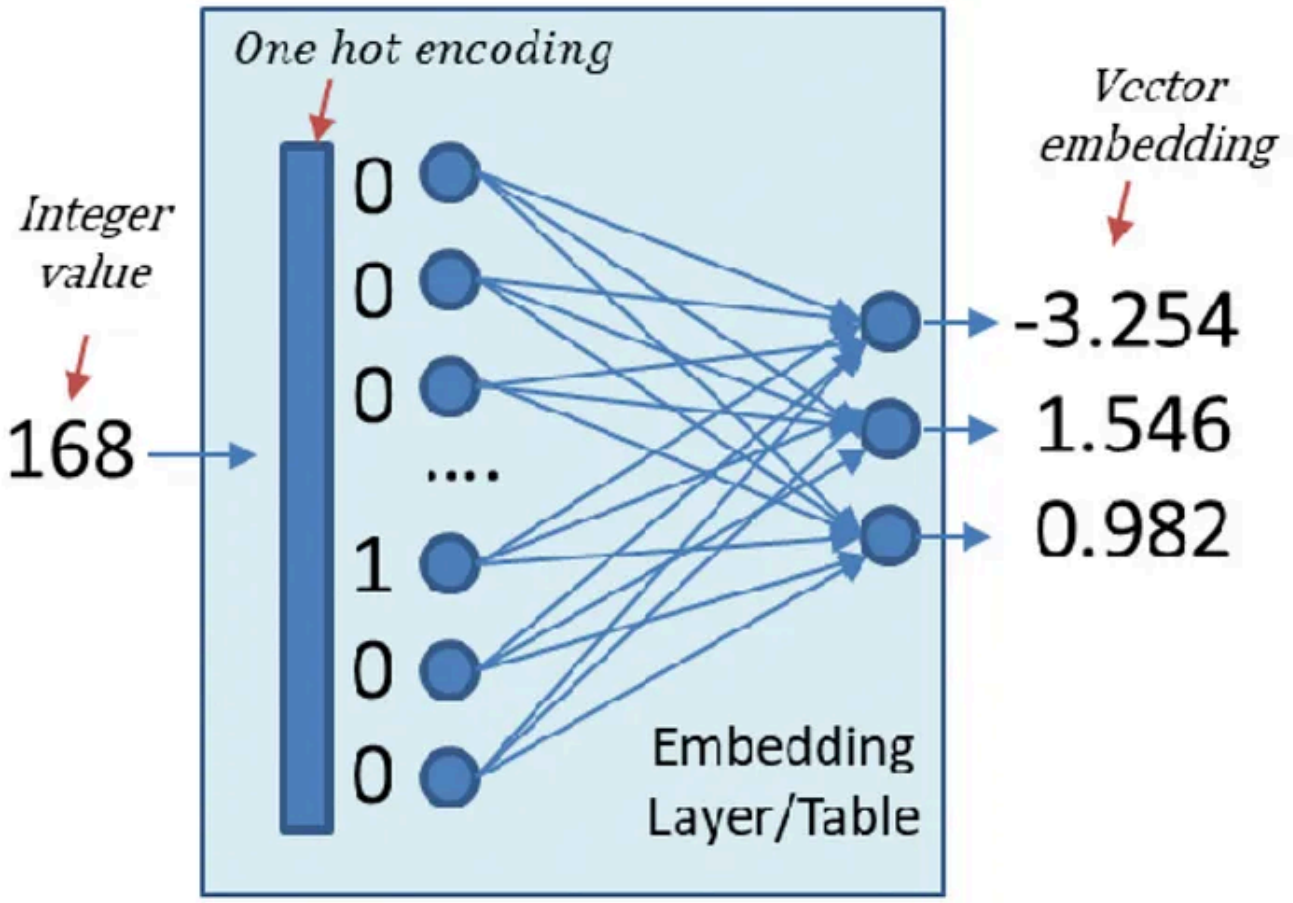


A 3 layer fully connected network in which dropout is applied at each layer.

It's crazy how effective this method works and it's a must for "parameter dense" layers such as linear layers that are prone to overfitting. It's actually not unusual to see drop out rates of 50% in some models. Obviously we want to avoid training a chess playing model that memorizes moves (there are too many to memorize!), but rather generalizes the training data and forces it to learn something closer to strategy.

Embedding Layer

An embedding layer is the first layer of a language model, responsible for converting each language token into an embedding vector. Since a token can be mapped directly to an ordinal value, an embedding layer is often implemented as just a lookup table, mapping a token directly to a vector.



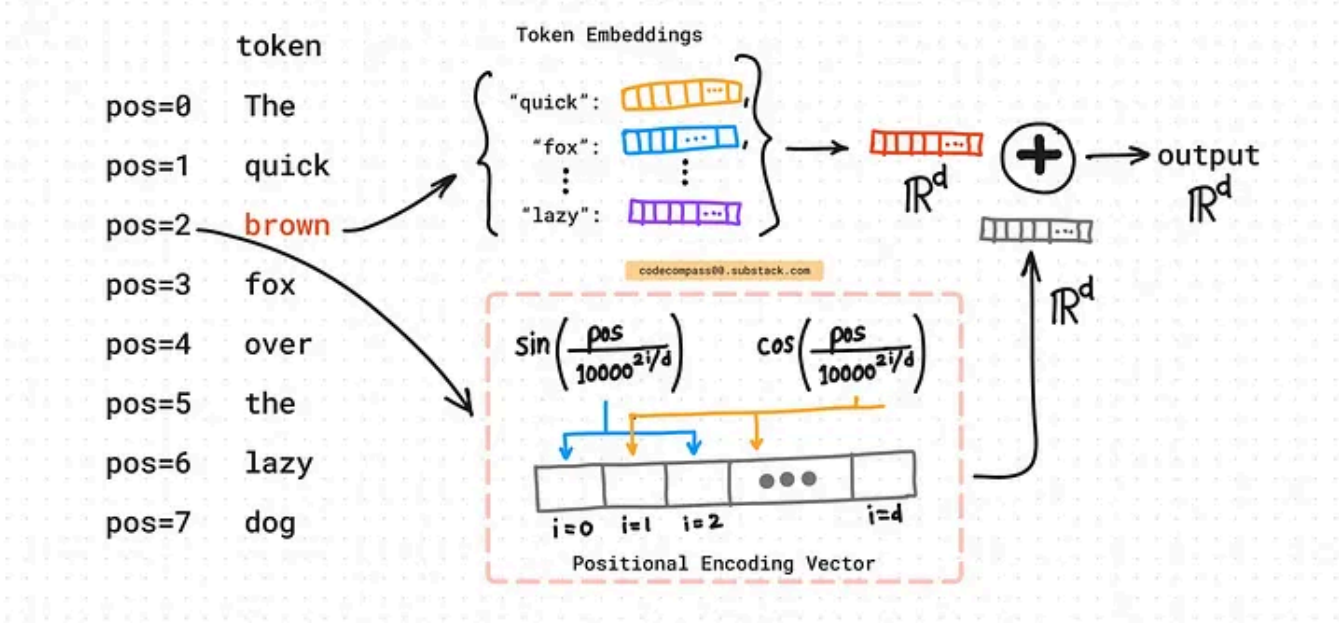
An embedding layer is just a lookup table. The lookup integer results in selecting a specific set of model weights which is the resulting embedding of the layer.

In many ways, a chess piece is just like a token or symbol. But instead of a sentence, it's positioned discretely on a chess board. I ended up using an embedding layer for my model for this reason.

Positional Encoding

You'll also find positional encoding in language models in addition to the embedding layer. The idea is simple. Consider the same token appearing in different positions in a sentence or paragraph. There needs to be a way to adjust that token's embeddings to reflect also its position. That's the role of position encoder.

The graphic below shows a simple scheme which adjusts the tokens embedding by the value of a trig function that's sensitive to the position.

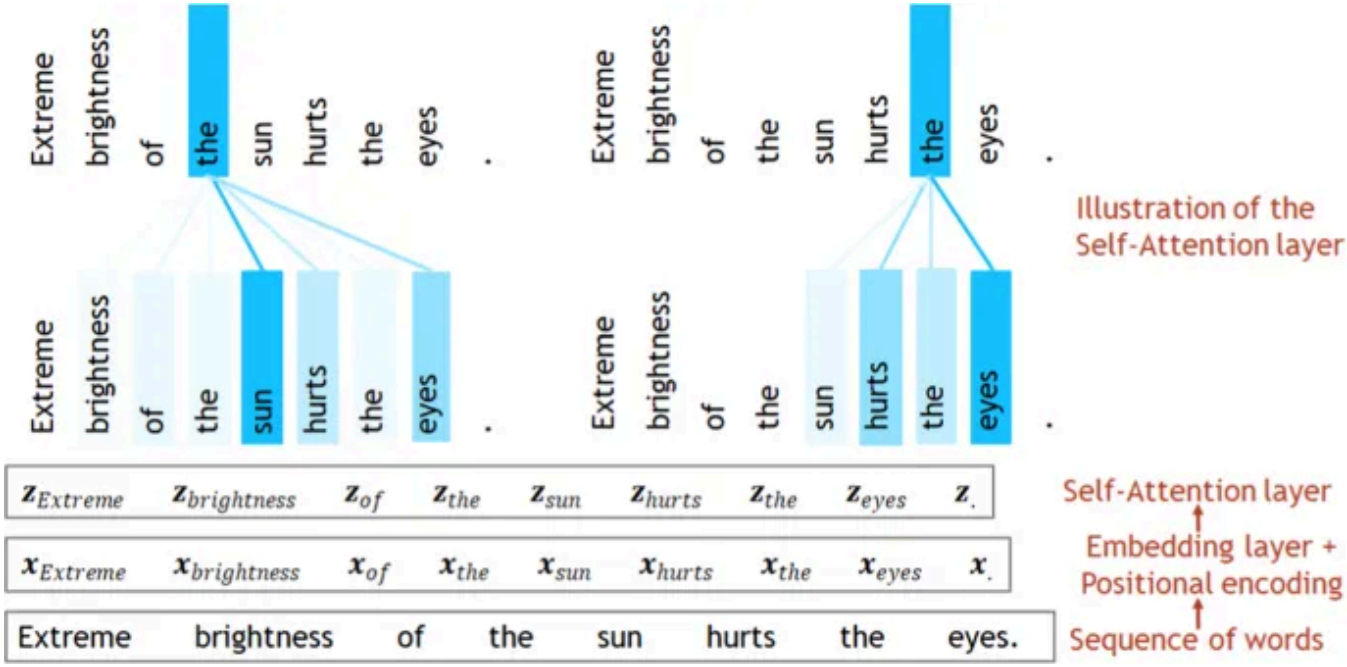


A position of the token produces a unique value of the trig function, which is added to the token embedding.

In a similar way, a chess piece (consider a pawn), can appear on different parts of a board. It could useful to encode a chess piece's position on the board just like what language models do with the same token in a sequence of text.

Attention Layer

The combination of embedding layer and positional encoder gives language models a powerful front-end that embeds each token at each position into the same vector space. The attention layers that follow do the heavy lifting of developing mathematical relationships in that shared vector space. The classic example of the attention mechanism shows how different words in a sentence are associated with the same word that appears in multiple positions of a sentence.



The word “the” appears twice in the sentence. Other words in the sentence “attend” to each appear of the word “the” in different ways.

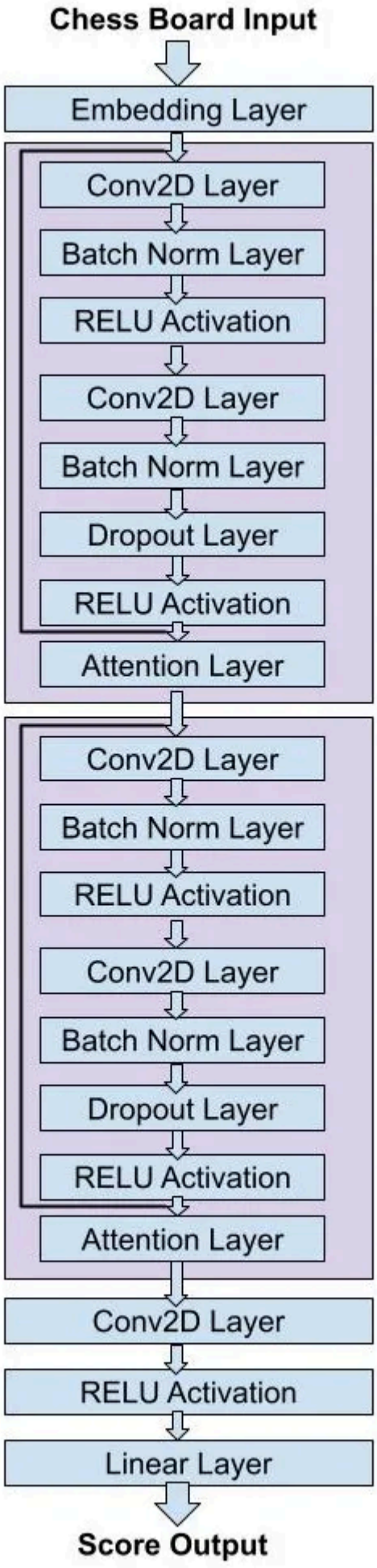
In the example above, the word “the” appears twice, but each appearance of “the” relates to different words in the sentence. The first “the” with “sun”, and the second “the” with “eyes.”

We might think about leveraging this attention mechanism to help a chess playing model associate chess pieces across the board. Could this attention mechanism help the model predict the best next move?

A Chess Playing Neural Network

During the hackathon, I explored several different architectures using the building blocks described above. In addition, Strong Compute provided several code implementations in PyTorch. They also gave participants access to a massive GPU cluster which encouraged experimentation with different model architectures.

Although participants could train lots of different models, the hackathon rules forced you to choose just one for the final submission. I ended up choosing the specific model shown below, because this one had trained for the longest period of time:



My winning hackathon chess playing model architecture!

A few things to note about this architecture:

- *Model Input and Output* The model takes as input a board state (not the entire game history) and predicts a score for that board. During tournament play, the game engine invokes the model with all potential board states associated with legal moves available for its turn. The game engine then chose the move with the highest score.
- *Residual Blocks* The purple block of layers is repeated twice in this architecture. I tried adding more blocks, but as I did so the size of the network started to exceed the resource limit restrictions set by the hackathon rules.
- *Residual Connection* Each purple block of layers has one residual connection shown as the bold black line connecting the block's input to the input of the final attention layer in the block. I enabled a model configuration setting which included learnable parameters in the residual connection, but I didn't have enough time to determine if it was useful or not during the hackathon. I ended up doing some experiments after the hackathon.

You can check out the model's PyTorch code here and the model configuration I used here. I'll get into more details of the model and the effect of different model configurations in the next blog.

What's Next?

Stay tuned for Part II in this series:

- I'll discuss details of training the model I described in this blog, including how I leveraged PyTorch's DDP (e.g., Distributed Data Parallel) to speed up training over several GPUs.
- I'll dive into the impact of different model configuration on training time and chess play performance. For example, after the hackathon I performed various "ablation studies" in which I trained models with slightly different tweaks, such as removing the attention layer, adding dilation, and adding more training data. I'll show the results of chess games in which I pitted these variants against each other.
- I'll get into the details of the Elo rating system, which is used for ranking human chess players and chess engines, based on their performance history.

If you liked this blog please show your love and hit the “like” button. And follow me on Medium to get a notification when I publish the next blog!

Chess Artificial Intelligence Machine Learning Hackathons Gpu

Follow

Written by George Williams

510 Followers · 1.3K Following

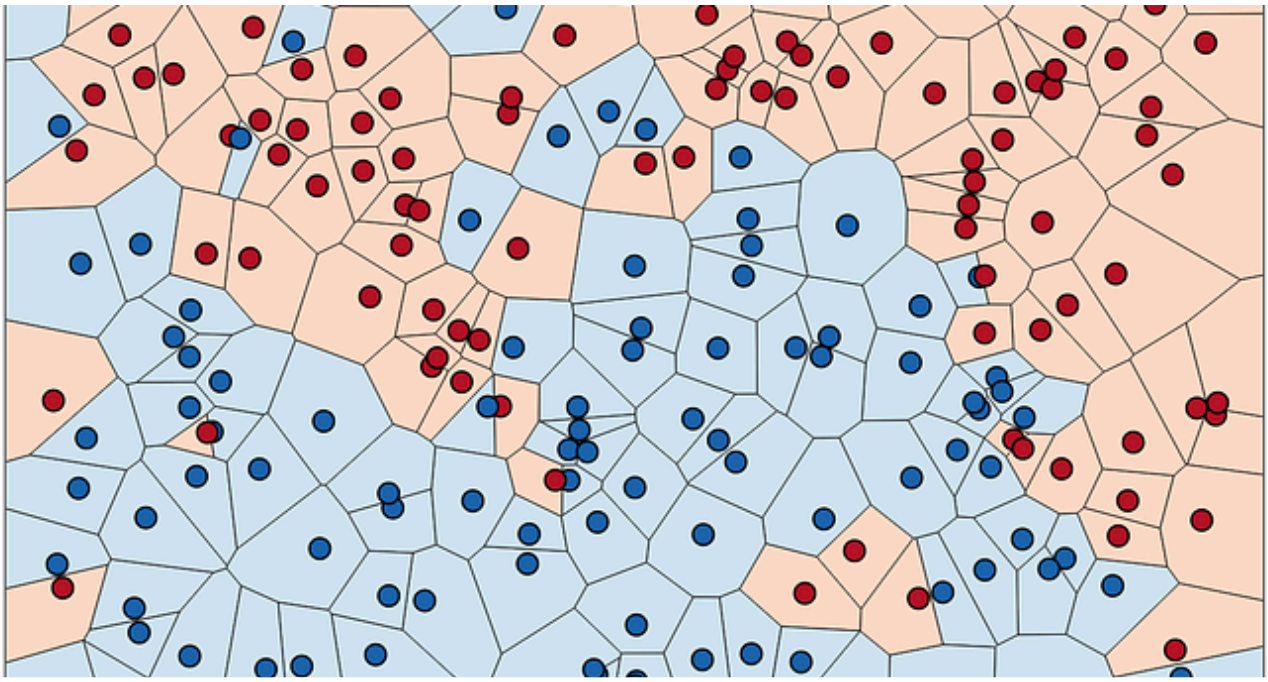
AI Consultant. Previously Head of AI at Smile Identity. Director of DS at GSIT, Chief DS at Sophos/Capsule 8, Senior DS at Apple, Motiv, Researcher at NYU.

No responses yet

What are your thoughts?

Respond

More from George Williams



In Big-ANN-Benchmarks by George Williams

NeurIPS 2021 Announcement: The Billion-Scale Approximate Nearest Neighbor Search Challenge

We are excited to announce that this year's NeurIPS 2021 Conference will host a first-of-its-kind competition in large scale approximate...

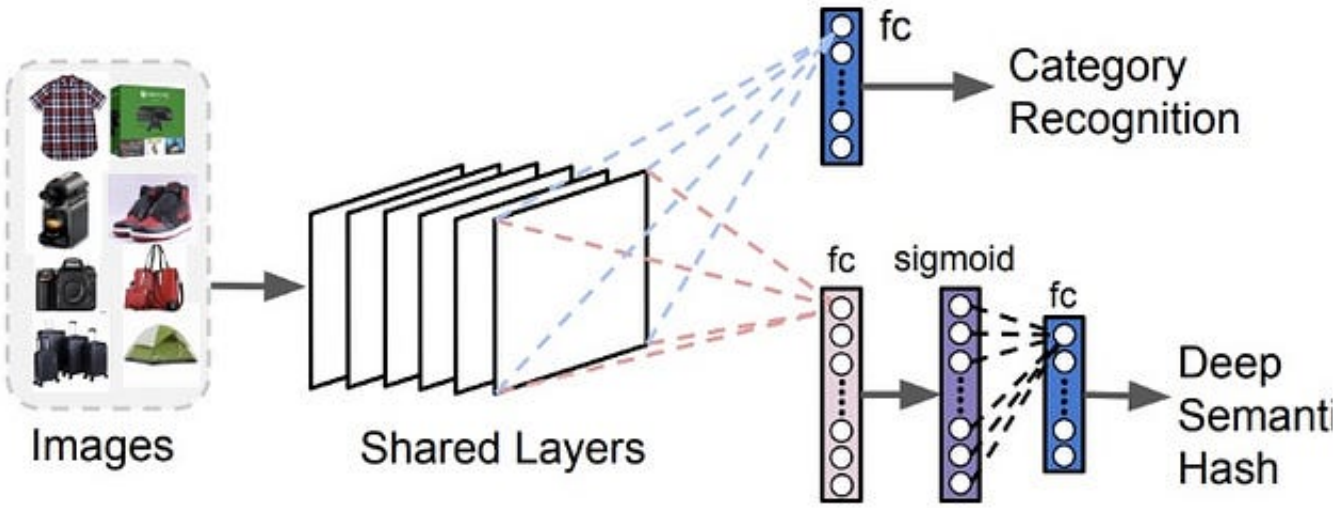
May 19, 2021 357

In Big-ANN-Benchmarks by George Williams

Watts The Deal With Power ? Part I

How We Implement Power Benchmarking In The Billion-Scale Approximate Nearest Neighbor Search Challenge

Jun 11, 2021 267



In GSI Technology by George Williams

ML In Visual Search

Online retailers piece together different kinds of technology to reduce friction and to keep us coming back as customers. From face...

Jun 16, 2018 433 1

In SmileIdentity by George Williams

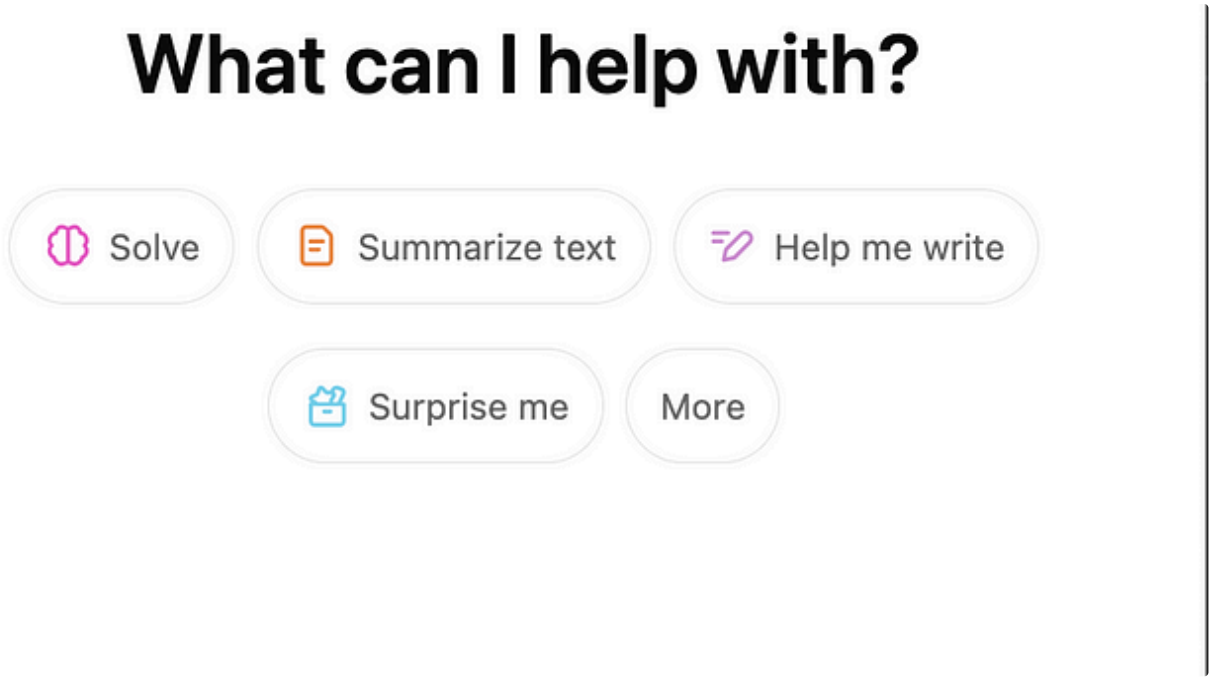
Not All Face Recognition Is Created Equal: The Need For Liveness.

Over the last two years of the covid-19 pandemic, the convenience that came with society's digital transformation has unfortunately been...

Apr 19, 2022 342

See all from George Williams

Recommended from Medium



Austin Starks

OpenAI just quietly released another agentic framework. It's really fucking cool

All of my articles are 100% free to read. Non-members can read for free by clicking my friend link here.

5d ago 882 25

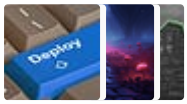
In Level Up Coding by Mark Lucking

The Art of Programming with AI

Using the Claude of Anthropic fame to build a SUDOKU game

3d ago 54

Lists



Predictive Modeling w/ Python

20 stories · 1819 saves

Natural Language Processing

1928 stories · 1583 saves

AI Regulation

6 stories · 685 saves

Practical Guides to Machine Learning

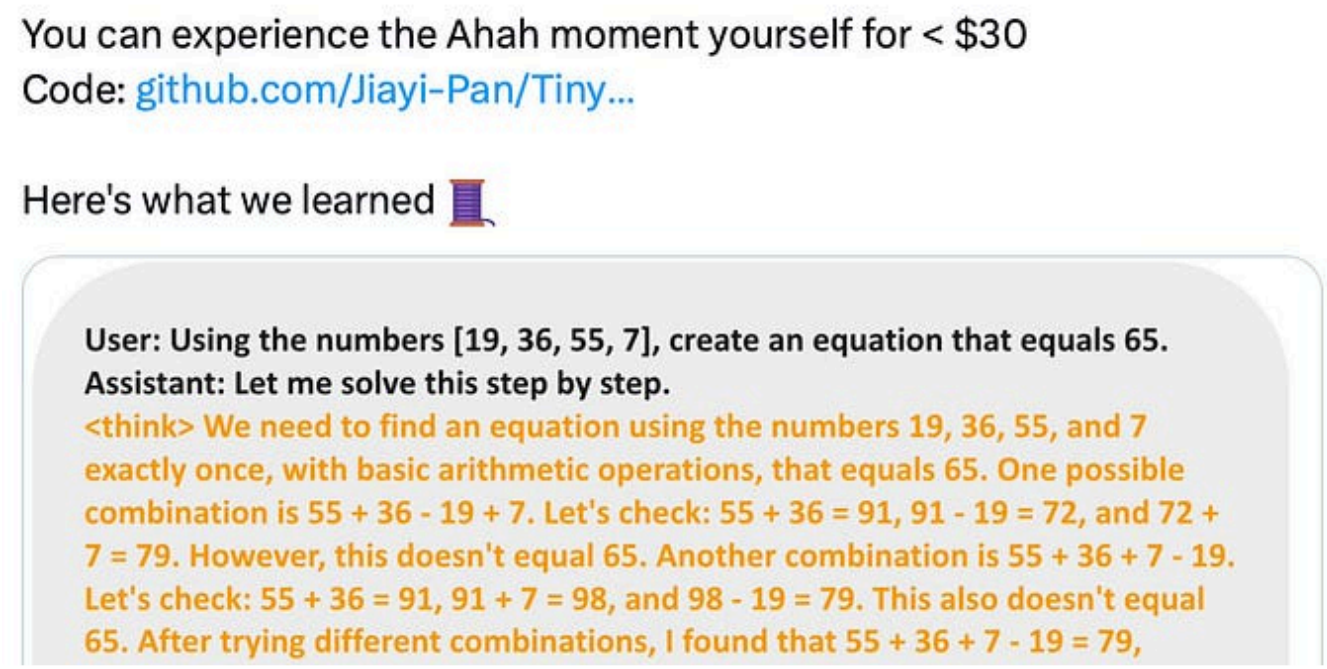
10 stories · 2189 saves

In Write A Catalyst by Onyedikachukwu Czar

DeepSeek Just Confirmed My Suspicions About OpenAI

The ChatGPT maker has been playing a losing game

Jan 28 3.1K 139

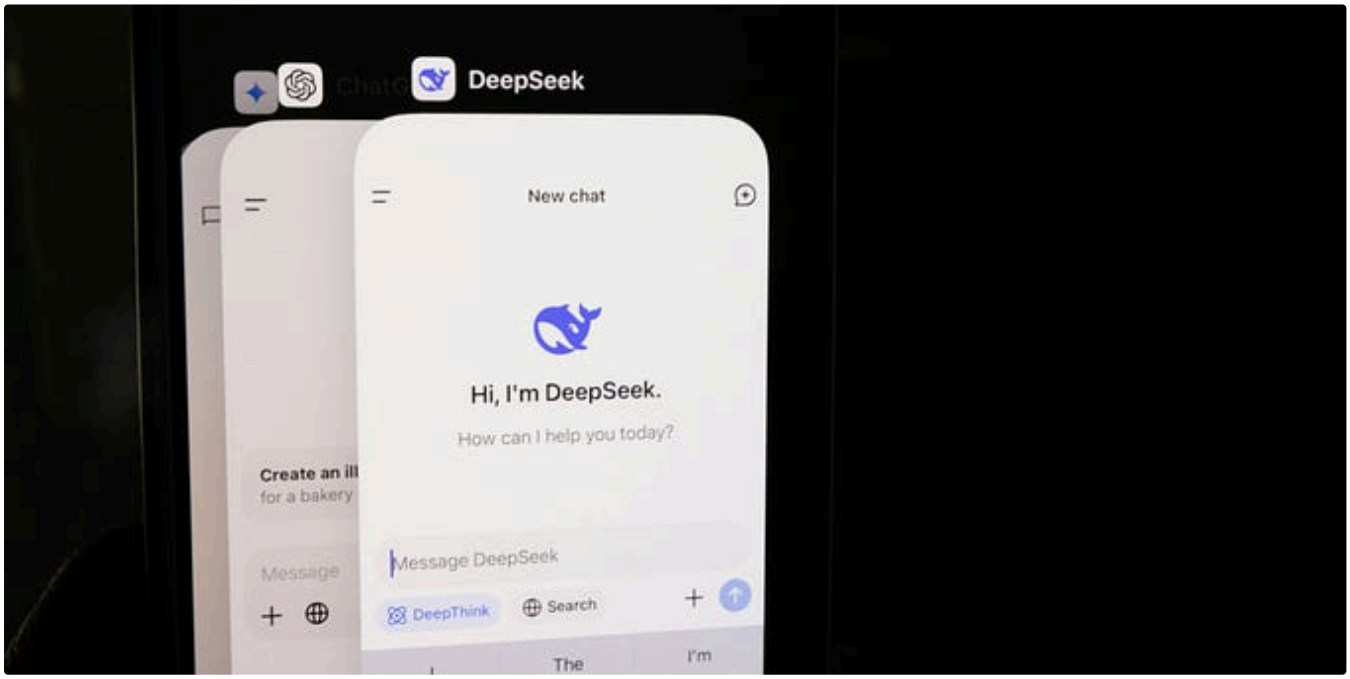


Wei Lu

How DeepSeek R1-Zero was reproduced in $30

Jiayi Pan, a PhD student at Berkeley, replicated DeepSeek R1-Zero's approach with just $30, enabling a 3B-parameter small model to achieve...

Jan 31 71 1



In Data Science in your pocket by Mehul Gupta

DeepSeek-R1 free API

How to use DeepSeek-R1 for free using OpenRouter

Feb 1 92 5

In Enrique Dans by Enrique Dans

With every move he makes, it's clearer than ever: Trump is going to crash the global economy

Believing that the rest of the planet is blind to his clumsy machinations, US president Donald Trump has signed an executive order to...

4d ago 4K 80

See more recommendations