Newton vs. The Machine

John Urbanic

Parallel Computing Scientist
Pittsburgh Supercomputing Center

Distinguished Service Professor
Carnegie Mellon University

But First:

Some Assignment #5 Solutions

Where to start?

Step 1: What do I mean by "better network"? This sounds like an actual research assignment! Well, if I run this

```
import tensorflow as tf

mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape(60000, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
train_images, test_images = train_images/255, test_images/255

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])


model.fit(train_images, train_labels, batch_size=32, epochs=10, verbose=1, validation_data=(test_images, test_labels))
```

We get this for our baseline.

```
Epoch 15/15
469/469 [==============================] - 1s 3ms/step - loss: 0.1213 - accuracy: 0.9555 - val_loss: 0.2384 - val_accuracy: 0.9284
```

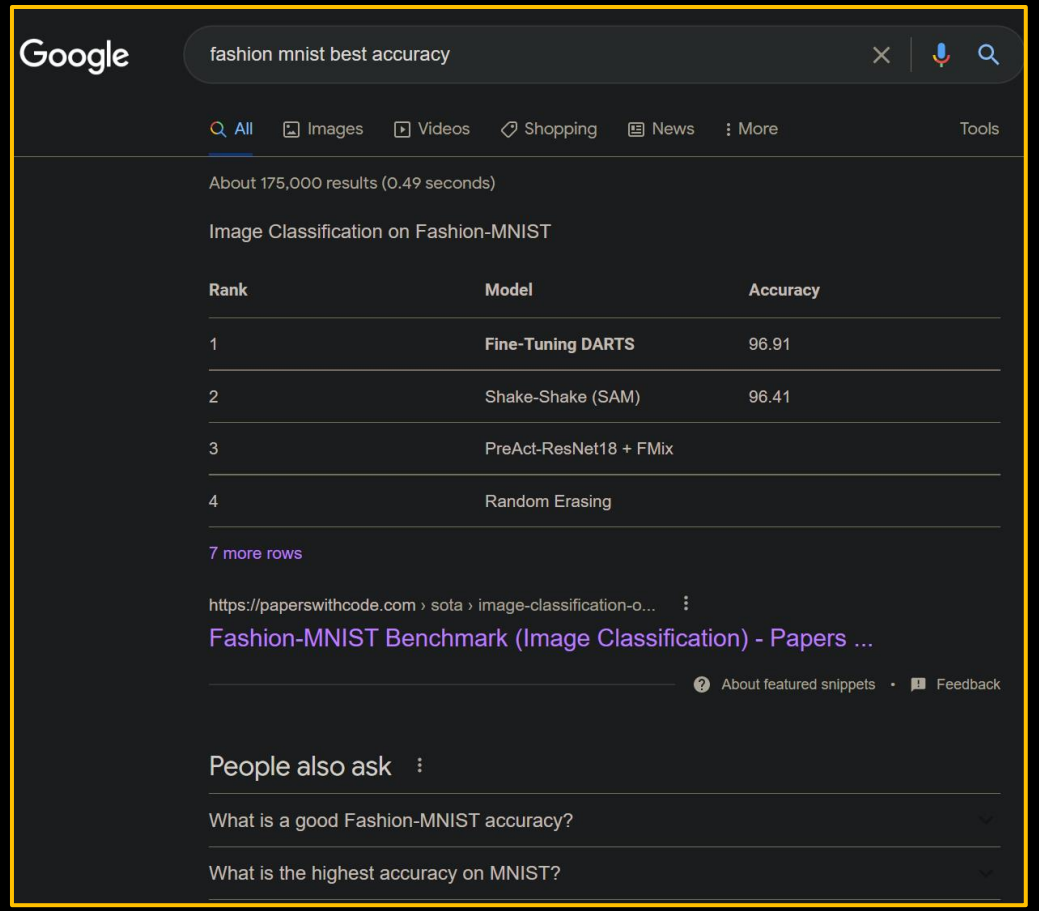
How far to go?

Step 2: As per my hint (and I hope your developing instincts), a good place to gain some perspective is always a little research.

It looks like some networks are getting 96%.

It also looks like those are some fancy networks.

Maybe I'll click a few more of those links...



Still researching.

This top link happens to provide a nice summary of the state-of-the-art in MNIST. That "kmnist" is a Japanese character version.

We know Alexnet is a big complex and powerful network, and yet it isn't doing as well as our baseline. This is a clue that bigger is not better.

Those other networks are also pretty big and fancy. So it looks like getting even 94% here is a big ask.

And we know I'm not an unreasonable fellow. So let's set our sights appropriately.

Dataset	Model	Best Epoch	Train Loss	Train Acc.	Val. Loss	Val. Acc.
kmnist	mobilenet_v2	89	0.036	99.988	0.095	98.468
kmnist	resnet101	90	0.020	99.993	0.097	98.147
kmnist	resnet14	73	0.021	99.975	0.070	98.748
kmnist	resnet152	85	0.020	99.988	0.090	98.197
kmnist	resnet18	73	0.020	99.997	0.077	98.628
kmnist	resnet34	83	0.018	99.998	0.081	98.538
kmnist	resnet50	80	0.020	99.982	0.097	98.067
kmnist	resnet9	54	0.008	99.997	0.069	98.528
kmnist	vgg11_bn	62	0.016	99.998	0.078	98.427
kmnist	vgg13_bn	54	0.016	99.980	0.069	98.698
kmnist	vgg16_bn	79	0.015	99.998	0.070	98.698
kmnist	vgg19_bn	99	0.015	99.998	0.078	98.518
fashionmnist	alexnet	97	0.296	89.248	0.308	89.042
fashionmnist	densenet121	92	0.037	99.947	0.266	93.950
fashionmnist	densenet161	93	0.038	99.937	0.264	94.171
fashionmnist	densenet169	97	0.038	99.933	0.258	94.291
fashionmnist	googlenet	96	0.044	99.973	0.240	93.439
fashionmnist	inception_v3	97	0.050	99.861	0.244	94.441
fashionmnist	mobilenet_v2	98	0.040	99.913	0.252	93.860
fashionmnist	resnet101	94	0.019	99.985	0.281	93.740
fashionmnist	resnet14	89	0.021	99.997	0.228	94.040
fashionmnist	resnet152	91	0.020	99.970	0.286	93.770
fashionmnist	resnet18	86	0.020	99.998	0.228	93.970
fashionmnist	resnet34	96	0.018	99.993	0.261	93.910
fashionmnist	resnet50	89	0.019	99.985	0.261	93.810
fashionmnist	resnet9	63	0.009	99.998	0.203	94.071
fashionmnist	vgg11_bn	91	0.017	100.000	0.229	93.600
fashionmnist	vgg13_bn	80	0.017	99.998	0.211	94.111
fashionmnist	vgg16_bn	86	0.018	99.995	0.226	94.030
fashionmnist	vgg19_bn	95	0.018	99.987	0.244	93.960

CNN Filter DB: An Empirical Investigation of Trained Convolutional Filters. Paul Gavrikov and Janis Keupe

Why so hard?

As per this analysis of the dataset, there are a lot debatable labels, and some that are apparently wrong.

This is just a trickier problem. Especially at this resolution.

I could not find the number for "superhuman" on this dataset, but it is clearly lower than for the digits.

Figure 7. Very hard or mislabelled thumbnails in Fashion-MNIST. A.T-shirt with person. B. Coat with person C .Dress with person. D. Dress with two parts. E .Dress with bloomers. F. T-shirts.

OVERHEAD MNIST: A BENCHMARK SATELLITE DATASET David A. Noever and Samantha E. Miller Noever

Get to work!

Step 3: There is no substitute for trial and error with the hyperparameters. Maybe we can hope to spot a trend.

This is a great example of the kind of approach to take.

This made it to over 94%.

Nesterov is just a fancier version of gradient descent (it "looks ahead") that you can select.

Change from Base Code (Acc = 0.8832)	Accuracy
Wider (512)	0.8927
Deeper (512,512,512)	0.8896
CNN (1 layer) - 100 Epochs	0.8790
CNN (2 layers) (32, 64)	0.9223
CNN (2 layers) (64, 64)	0.9217
Drop (0.25,0.5) - 100 Epochs	0.9322
Drop (0.5,0.5) - 80 Epochs	0.9338
Batch Normalization	0.9279
Changing Activation Function (“sigmoid”)	0.8678
Conv2D strides = (5,5), lr = 0.02, momentum = 0.9	0.9252
Conv2D strides = (3,3), lr = 0.02, momentum = 0.8	0.9256
lr = 0.02, momentum = 0.9	0.9301
Changing (padding="same") - 40 epochs	0.9364
Changing (padding="same") - 80 epochs	0.9372
nesterov=True - 40 epochs	0.9362
maxPooling(padding=“same”)	0.9359
nesterov=False	0.9334
nesterov=False - 120 epochs	0.9381
Dropout (0.5,0.6), Epochs = 40	0.9314 (Max: 0.9388)
Dropout (0.6,0.6), Epochs = 40	0.9356
Dropout (0.75,0.75), Epochs = 40	0.9266
Dropout (0.2,0.6), Epochs = 40	0.9336
Dropout (0.2,0.6), Epochs = 100	0.9355
Dropout (0.6,0.6), Epochs = 100	0.9384 (Max: 0.9422)
nesterov=True, Epochs = 100	0.9418

Another fine pursuit of good hyperparameters.

93.7% and a systematic two-steps-forward-one-step-back attack.

#	Model	Train	Test
1	Baseline w/ Dropout (trained for 20 epochs)	0.9427	0.918
2	Changed optimizer to ADAM	0.9619	0.9267
3	Increase first conv layer to 64 filters	0.9612	0.9315
4	Increase second conv layer to 128 filters	0.968	0.9303
5	Increase dense layer to 192	0.9746	0.9335
6	Increase dense layer to 256	0.9814	0.9264
7	Increase dropout after conv layer to 0.4	0.971	0.932
8	Increase dropout after conv layer to 0.5	0.9627	0.9317
9	Add conv layer with 96 filters between existing conv layers	0.9634	0.9318
10	Removed new layer, added third conv layer with 192 filters	0.9667	0.9343
11	Add dense layer with 256 neurons, dropout 0.5	0.9467	0.931
12	Reduce dropout of new layer to 0.25	0.9653	0.9308
13	Remove dropout of new layer	0.9817	0.9256
14	Change new layer to have 320 neurons, reintroduce 0.25 dropout	0.9702	0.9303
15	Revert to model 10, changed third conv layer to have 224 filters	0.9648	0.9348
16	Revert to model 10, changed second conv layer to have 192 filters	0.9626	0.9356
17	Changed first conv layer to have 128 filters	0.9655	0.932
18	Revert to model 16, increase dense layer to have 320 neurons	0.9677	0.9293
19	Revert to model 16, train for 30 epochs	0.9688	0.9338
20	Changed third conv layer to have 256 filters, train for 25 epochs	0.9691	0.9303
21	Changed dense layer to have 320 neurons	0.9733	0.9331
22	Changed dense layer to have 512 neurons	0.9749	0.9357
23	Add BatchNorm after all conv layers before pooling	0.9691	0.9316
24	Add first dense layer with 1024 neurons, dropout 0.5	0.952	0.9342
25	Add conv layer with 64 filters and BatchNorm after first layer	0.948	0.9335
26	Revert to model 16, add BatchNorm after all conv layers before pooling	0.9654	0.9368

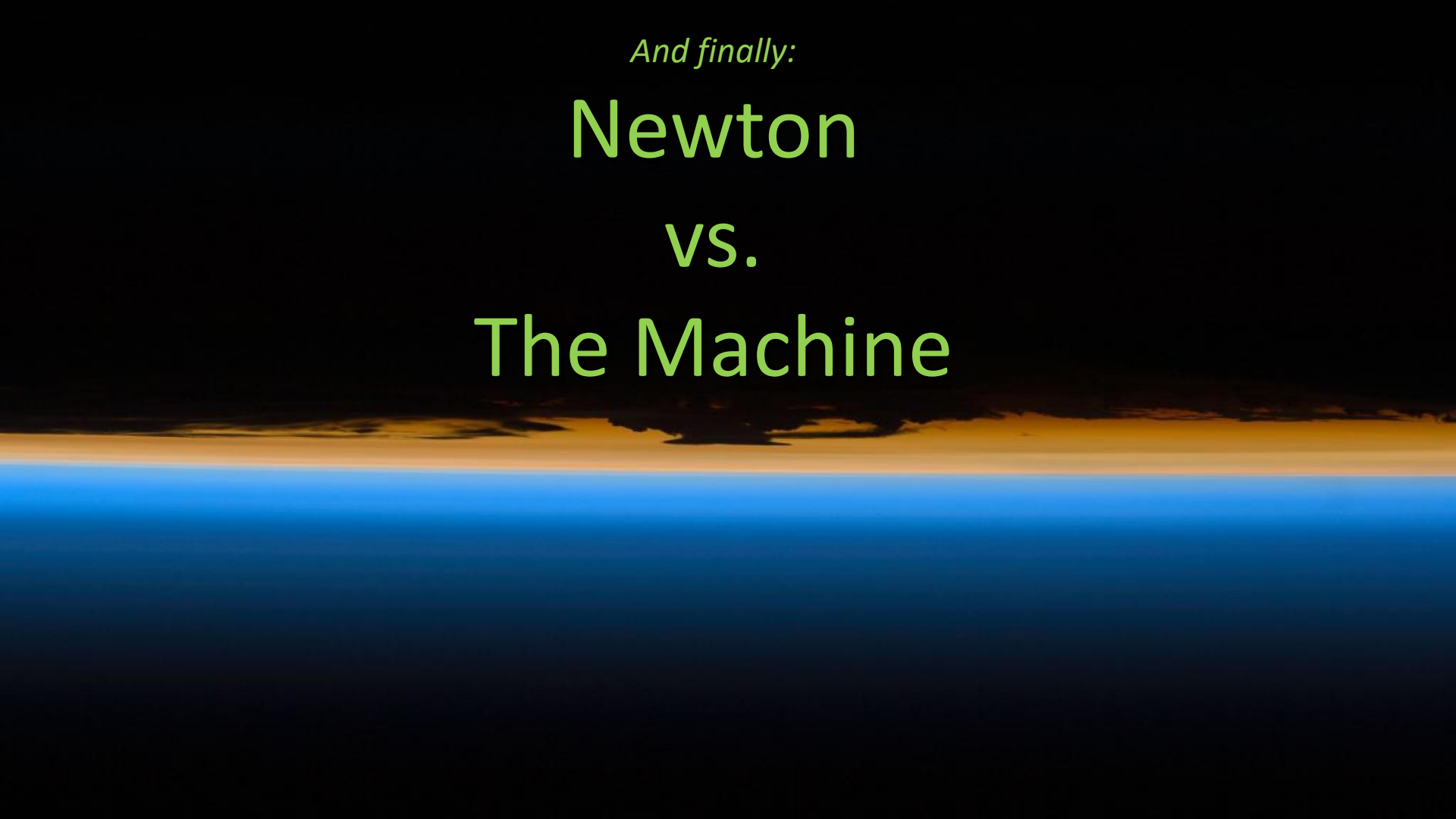
Early Stopping

Of course, manual monitoring is not he only way to prevent needless training after the loss has plateaued. There are numerous ways in any framework to enable early stopping.

The most common is to add the *EarlyStopping* callback to your training loop.

```
tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    min_delta=0,
    patience=0,
    verbose=0,
    mode='auto',
    baseline=None,
    restore_best_weights=False,
    start_from_epoch=0
)
```

`monitor`	Quantity to be monitored. Defaults to `"val_loss"`.
`min_delta`	Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement. Defaults to `0`.
`patience`	Number of epochs with no improvement after which training will be stopped. Defaults to `0`.
`verbose`	Verbosity mode, 0 or 1. Mode 0 is silent, and mode 1 displays messages when the callback takes an action. Defaults to `0`.
`mode`	One of `{"auto", "min", "max"}`. In `min` mode, training will stop when the quantity monitored has stopped decreasing; in `"max"` mode it will stop when the quantity monitored has stopped increasing; in `"auto"` mode, the direction is automatically inferred from the name of the monitored quantity. Defaults to `"auto"`.
`baseline`	Baseline value for the monitored quantity. If not `None`, training will stop if the model doesn't show improvement over the baseline. Defaults to `None`.
`restore_best_weights`	Whether to restore model weights from the epoch with the best value of the monitored quantity. If `False`, the model weights obtained at the last step of training are used. An epoch will be restored regardless of the performance relative to the `baseline`. If no epoch improves on `baseline`, training will run for `patience` epochs and restore weights from the best epoch in that set. Defaults to `False`.
`start_from_epoch`	Number of epochs to wait before starting to monitor improvement. This allows for a warm-up period in which no improvement is expected and thus training will not be stopped. Defaults to `0`.


And finally:
Newton
vs.
The Machine

MNRAS 000, 1–6 (2019) Preprint 17 October 2019 Compiled using MNRAS LATEX style file v3.0

Newton vs the machine: solving the chaotic three-body problem using deep neural networks

Philip G. Breen[1]★†, Christopher N. Foley[2] ★‡, Tjarda Boekholt[3] and Simon Portegies Zwart[4]

[1]School of Mathematics and Maxwell Institute for Mathematical Sciences, University of Edinburgh, Kings Buildings, Edinburgh, EH9 3JZ
[2]MRC Biostatistics Unit, University of Cambridge, Cambridge, CB2 0SR, UK.
[3]Instituto de Telecomunicações, Campus Universitário de Santiago, 3810-193, Aveiro, Portugal
[4]Leiden Observatory, Leiden University, PO Box 9513, 2300 RA, Leiden, The Netherlands.

Accepted XXX. Received YYY; in original form ZZZ

ABSTRACT

Since its formulation by Sir Isaac Newton, the problem of solving the equations of motion for three bodies under their own gravitational force has remained practically unsolved. Currently, the solution for a given initialization can only be found by performing laborious iterative calculations that have unpredictable and potentially infinite computational cost, due to the system's chaotic nature. We show that an ensemble of solutions obtained using an arbitrarily precise numerical integrator can be used to train a deep artificial neural network (ANN) that, over a bounded time interval, provides accurate solutions at fixed computational cost and up to 100 million times faster than a state-of-the-art solver. Our results provide evidence that, for computationally challenging regions of phase-space, a trained ANN can replace existing numerical solvers, enabling fast and scalable simulations of many-body systems to shed light on outstanding phenomena such as the formation of black-hole binary systems or the origin of the core collapse in dense star clusters.

Key words: stars: kinematics and dynamics, methods: numerical, statistical

1 INTRODUCTION

Newton's equations of motion describe the evolution of many bodies in space under the influence of their own gravitational force (Newton 1687). The equations have a central role in many classical problems in Physics. For example, the equations explain the dynamical evolution of globular star clusters and galactic nuclei, which are thought to be the production sites of tight black-hole binaries that ultimately merge to produce gravitational waves (Portegies Zwart & McMillan 2000). The fate of these systems depends crucially on the three-body interactions between black-hole binaries and single black-holes (e.g. see Breen & Heggie 2013A,B; Samsing & D'Orazio 2018), often referred to as close encounters. These events typically occur over a fixed time interval and, owing to the tight interactions between the three nearby bodies, the background influence of the other bodies can be ignored, i.e. the trajectories of three bodies can be generally computed in isolation (Portegies Zwart & McMillan 2018). The focus of the present study is therefore the timely computation of accurate solutions to the three-body problem.

Despite its age and interest from numerous distinguished scientists (de Lagrange 1772; Heggie 1975; Hut & Bahcall 1983; Montgomery 1998; Stone & Leigh 2019), the problem of solving the equations of motion for three-bodies remains impenetrable due to the system's chaotic nature (Valtonen et al 2016) which typically renders identification of solutions feasible only through laborious numerical integration. Analytic solutions exist for several special cases (de Lagrange 1772) and a solution to the problem for all time has been proposed (Valtonen et al 2016), but this is based on an infinite series expansion and has limited use in practice. Computation of a numerical solution, however, can require holding an exponentially growing number of decimal places in memory and using a time-step that approaches zero (Boekholt et al 2019). Integrators which do not allow for this often fail spectacularly, meaning that a single numerical solution is unreliable whereas the average of an ensemble of numerical solutions appear valid in a statistical sense, a concept referred to as nagh Hoch (Portegies Zwart & Boekholt 2018). To overcome these issues, the Brutus integrator was developed (Boekholt & Portegies Zwart 2015),

★ Authors contributed equally
† Contact e-mail: phil.breen@ed.ac.uk
‡ Contact e-mail: christopher.foley@mrc-bsu.cam.ac.uk

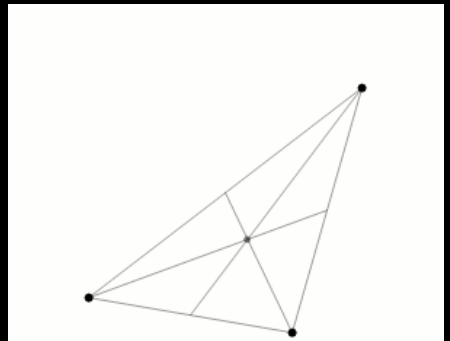
The Three-body Problem

You may have studied the two-body problem. One of its characteristics is that it is wonderfully analyzable for a variety of potentials.

The next step up the ladder of n-body problems is the three-body model. This has been an active area of research since Newton as it not only applies to such important cases as the Sun-Earth-Moon, but it also defies simple analysis.

We know that the general solution can not be represented by any closed solution (although there has been relatively recent work regarding infinite series).

Many of the greatest mathematicians have devoted efforts to the problem and some particular solutions were found by Newton, Jacob Bernoulli, Daniel Bernoulli, Leonhard Euler, Joseph-Louis Lagrange, Pierre-Simon Laplace, and others. Henri Poincaré's efforts were a large part of the foundation of modern chaos theory.



Wikipedia Commons

Numerically Challenging

Generating the training data here exemplifies how difficult it is to do this accurately. The "classical" method that they used (actually quite sophisticated, and a very modern code) uses an advanced numerical method and includes variable precision.

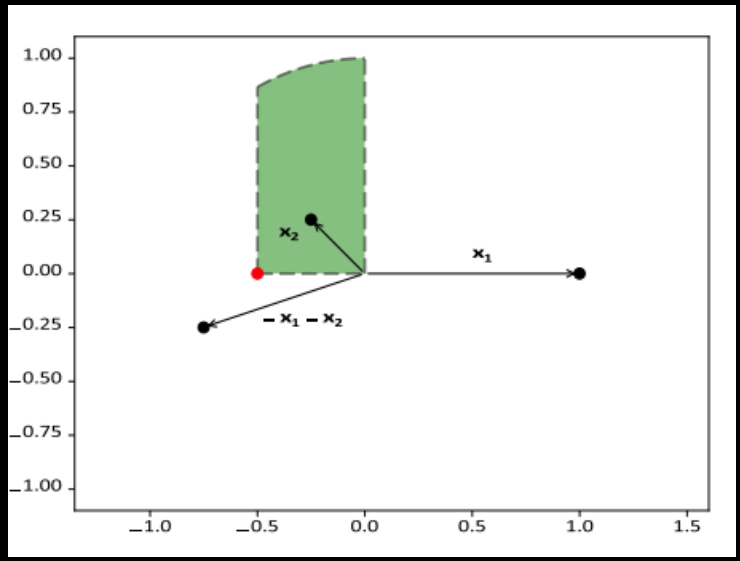
Despite its age and interest from numerous distinguished scientists (de Lagrange 1772; Heggie 1975; Hut & Bahcall 1983; Montgomery 1998; Stone & Leigh 2019), the problem of solving the equations of motion for three-bodies remains impenetrable due to the system's chaotic nature (Valtonen et al 2016) which typically renders identification of solutions feasible only through laborious numerical integration. Analytic solutions exist for several special cases (de Lagrange 1772) and a solution to the problem for all time has been proposed (Valtonen et al 2016), but this is based on an infinite series expansion and has limited use in practice. Computation of a numerical solution, however, can require holding an exponentially growing number of decimal places in memory and using a time-step that approaches zero (Boekholt et al 2019). Integrators which do not allow for this often fail spectacularly, meaning that a single numerical solution is unreliable whereas the average of an ensemble of numerical solutions appear valid in a statistical sense, a concept referred to as nagh Hoch (Portegies Zwart & Boekholt 2018). To overcome these issues, the Brutus integrator was developed (Boekholt & Portegies Zwart 2015), allowing for close-to-zero time-steps and arbitrary precision. Brutus is capable of computing converged solutions to any gravitational N-body problem, however the process is laborious and can be extremely prohibitive in terms of computer time. In general, there does not exist a theoretical framework capable of determining a priori the precision required to deduce that a numerical solution has converged for an arbitrary initialization (Stone & Leigh 2019). This makes the expense of acquiring a converged solution through brute-force integration unpredictable and regularly impractical.

A converged solution was acquired by iteratively reducing two parameters during integration: (i) the tolerance parameter (ϵ), controlling accuracy, that accepts convergence of the Bulirsch-Stoer multi-step integration scheme (Bulirsch & Stoer 1964) and; (ii) the word length (L_w) measured in bits, which controls numerical precision (Boekholt & Portegies Zwart 2015). Our ensemble of initial realizations all converged for values of $\epsilon = 10^{-11}$ and $L_w = 128$ (see Appendix A). Generating these data required over 10 days of computer time. Some initialisations gave rise to very close

Reducing the Parameters

For this restricted version of the problem (3 equal masses with zero initial velocities) we can use a little physics and reduce the number of parameters required to represent the initial configurations.

- Reference frame is center of mass
- X_1 is always at (1,0)
- Specify X_2's x,y coordinates
- X_3 is given by symmetry ($-X_1-X_2$)



We only need two parameters (X_2 x,y coordinates) to specify any desired initial condition!

If you recall our "sophisticated" view of neural nets as mappings from input vectors to output vectors, you can imagine we have simplified the task for our network a good deal.

Training Data

Our training data is going to have an input file with each sample consisting of an initial configuration and a timestamp for the desired final configuration.

The corresponding output is going to be the numerically generated final configuration in a reduced form: X_1 and X_2 positions (X_3 follows from symmetry).

train_X.csv

```
0.652344,-0.1648881236291216,0.9141665763358605
0.519531,-0.1648881236291216,0.9141665763358605
0.21875,-0.1648881236291216,0.9141665763358605
...
```

train_Y.csv

```
0.8724009514007633,0.03988576497011263,-0.1041928066877228,0.7964733750758993
0.9205503757602744,0.02465380676629187,-0.127317806825031,0.8407822474813996
0.9862529823431486,0.004226939675911646,-0.1584345882281039,0.9014472881328324
...
```

Our dataset will only contain the <3.9s class of data - or about 39% of the longest, 10s, trajectories mentioned in the paper.

Your Model

The paper describes the network used to generate the results. In particular, all of the hyperparameters.

This is everything you need to know to reproduce their results.

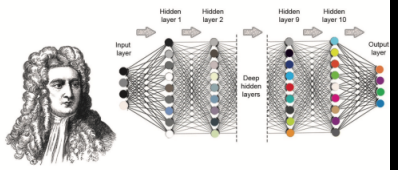


Figure 2. Newton and the machine. Image of sir Isaac Newton alongside a schematic of a 10-layer deep neural network. In each layer (apart from the input layer), a node takes the weighted input from the previous layer's nodes (plus a bias) and then applies an activation function before passing data to the next node. The weights (and bias) are free parameters which are updated during training.

counters, and computation of converged solutions in these situations is costly[1] (Boekholt et al 2019).

We used a feed-forward ANN consisting of 10 hidden layers of 128 interconnected nodes (Fig. 2 and Appendix B). Training was performed using the adaptive moment estimation optimization algorithm ADAM (20) with 10000 passes over the data, in which each epoch was separated into batches of 5000, and setting the rectified linear unit (ReLU) activation function to $\max(0, x)$ (Glorot, Bordes & Bengio, 2011). By entering a time t and the initial location of particle x_2 into the input layer, the ANN returns the locations of the particles x_1 and x_2 at time t, thereby approximating the latent analytical solution to the general three-body problem.

To assess performance of the trained ANN across a range of time intervals, we partitioned the training and validation datasets into three segments: $t \lesssim 3.9$, $t \lesssim 7.8$ and $t \lesssim 10$ (which includes all data). For each scenario, we assessed the loss-function (taken as the mean absolute error MAE) against epoch. Examples are given in Fig. 3. In all scenarios the loss in the validation set closely follows the loss in the training set. We also assessed sensitivity to the choice of activation function, however no appreciable improvement was obtained when using either the exponential rectified (Clevert, Unterthiner & Hochreiter 2011) or leaky rectified (Maas, Hannun & Ng 2013) linear unit functions. In addition, we assessed the performance of other optimization schemes for training the ANN, namely an adaptive gradient algorithm (Duchi, Hazan & Singer 2011) and a stochastic gradient descent method using Nesterov momentum, but these regularly failed to match the performance of the ADAM optimizer.

Results

You should be able to reproduce these same results for the <3.9s data.

You should include an error graph that looks very similar to theirs.

Reproducing some sample trajectories is optional, but recommended to make sure you really understand what you are doing. *This is worth a lot of partial credit if you need it!*

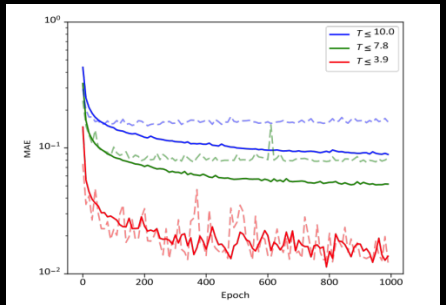


Figure 3. Mean Absolute Error (MAE) vs epoch. The ANN has the same training structure in each time interval. Solids lines are the loss on the training set and dashed are the loss on the validation set. $T \leq 3.9$ corresponds to 1000 labels per simulation, similarly $T \leq 7.8$ to 2000 labels and $T \leq 10.0$ to 2561 labels/time-points (the entire dataset).The results illustrate a typical occurrence in ANN training, there is an initial phase of rapid learning, e.g. âĽš100 epochs, followed by a stage of much slower learning in which relative prediction gains are smaller with each epoch.

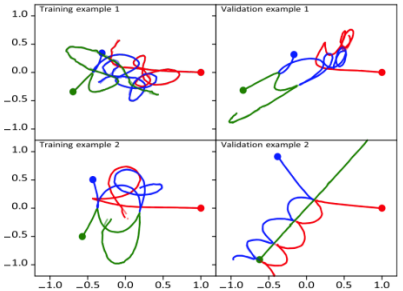


Figure 4. Validation of the trained ANN. Presented are two examples from the training set (left) and two from the validation set (right). All examples were randomly chosen from their datasets. The bullets indicate the initial conditions. The curves represent the orbits of the three bodies (red, blue and green, the latter obtained from symmetry). The solution from the trained network (solid curves) is hardly distinguishable from the converged solutions (dashes, acquired using Brutus (Boekholt & Portegies Zwart 2015)). The two scenarios presented to the right were not included in the training dataset.

Chaotic Motion

Chaotic motions like these are not only interesting and challenging, but they represent a great deal of everyday mechanics, not just double-pendulums. Most fluid dynamics (meteorology, etc.) is firmly in this category.

10 years ago we might have spent considerable time on this topic. Alas, it has been displaced by all of our very modern techniques. The good news is that these very modern techniques have given us exciting new tools for these classic problems.

Some flavor of the central characteristic of chaotic motion - a sensitivity to initial conditions - is explored here. Some very basic insight into how the trajectories diverge with consistency is seen here.

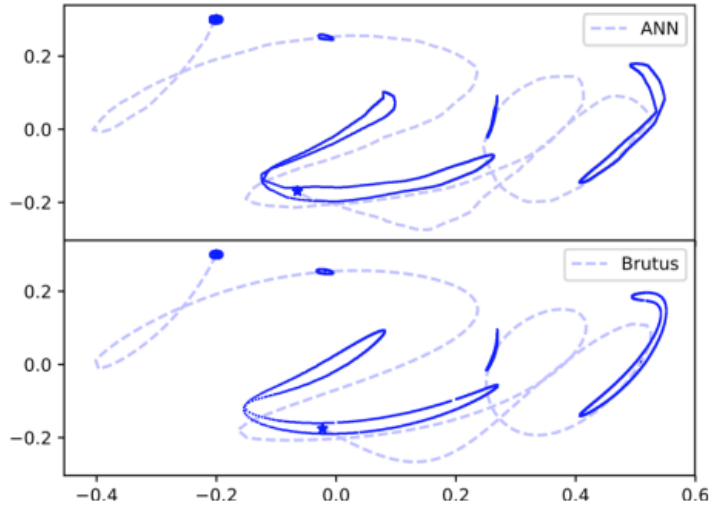


Figure 5. Visualization of the sensitive dependence on initial position. Presented are trajectories from 1000 random initializations in which particle x_2 is initially situated on the circumference of a ring of radius 0.01 centred at (-0.2, 0.3). For clarity, these locations were benchmarked against the trajectory of x_2 initially located at the centre of the ring (hatched line), the star denotes the end of this trajectory after 3.8 time-units. None of these trajectories were part of the training or validation datasets. The locations of the particles at each of five timepoints, $t \in \{0.0, 0.95, 1.9, 2.95, 3.8\}$, are computed using either the trained ANN (top) or Brutus (bottom) and these solutions are denoted by the bold line. The results from both methods closely match one another and illustrate a complex temporal relationship which underpins the growth in deviations between particle trajectories, owing to a change in the initial position of x_2 on the ring.

Conserved Quantities

Even complex systems may have conserved quantities. Ours certainly does, and it makes a great check of correctness.

Our network doesn't actually output velocities directly. The most obvious thing to do would be to derive these using the trajectories.

The paper discusses a network that actually generates velocities. It also discusses an improved algorithm that incorporates this into an improved error term.

There remain interesting areas of improvement to explore. That you could do!

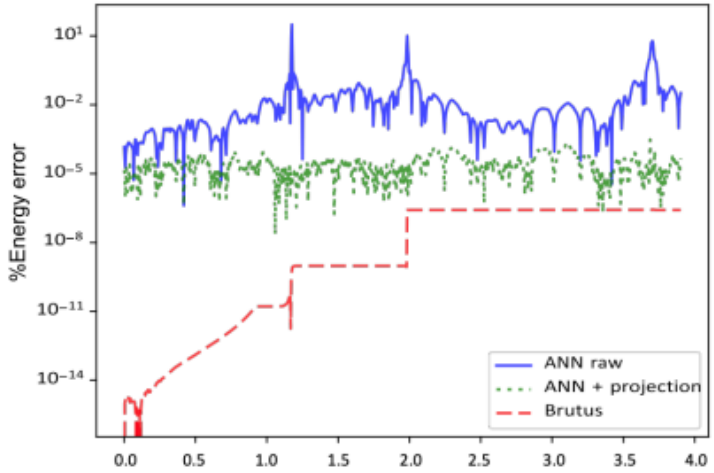


Figure 6. Relative energy error. An example of the relative energy error for a typical simulation. The raw output of the two ANN's typically have errors of around 10^{-2}, after projecting onto a near by energy surface the error reduces down to order 10^{-5}.

Assignment #6: Newton vs. The Machine

We are only concerned with the most accurate trajectories, the case of t < 3.9s. Otherwise the dataset is as described in the paper*.

The files are found at `~urbanic/LargeScaleComputing/Newton_vs_Machine`, along with a copy of the paper.

In this format, the data can be easily read. One way to get it into a TensorFlow compatible numpy array is with

```
import pandas as pd
inputf = pd.read_csv("train_X.csv").values
target = pd.read_csv("train_Y.csv").values
```

but feel free to do something else.

I used the AI module for my own tests (as opposed to the singularity container I recommended for the MNIST work). Either should work, but you can use the AI module with a simple

```
module load AI
```

and then just start your python shell or run your python script and TensorFlow will be there.

* *We did eliminate two corrupted trajectories. You won't notice unless you count all the samples.*

Assignment #6: Newton vs. The Machine

To reproduce the results shown in Figure 3 of the paper (for t < 3.9s) you will need to run 1000 epochs. We found this to take about 100 minutes.

The good news is that you shouldn't be doing much of a hyperparameter search as the paper is handing you all of the effective values. If you nail it, you will only need one run.

I don't expect you to nail it the first time. So, pay attention and if the error isn't looking something like Figure 3, kill your run and try again.

The paper has the (very small) validation split of only 1% of the data. You can split this out of the given data yourself (use Spark!). Or, this might be a good time to learn about the `validation_split` option in `model.fit()`.

Assignment #6: Logistics

Deadline: December 5th, last day of class. Before we start the lecture.

Points: 10 0points.

Format: Email me your TensorFlow script as an attachment and the detailed results in the body of the email. This includes at least the final test accuracy and the error graph. You may also want to include comments as your incremental efforts along the way will count for credit.

Course Evaluations

Course Evaluations just came online.

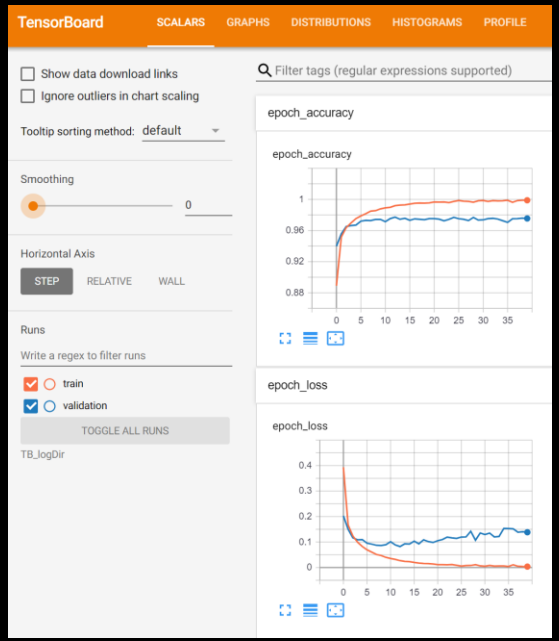
This course is under very active development, with no real peers. Your feedback is critical to help us decide if we continue in this direction.

Please take a few moments and let us know what you thought.

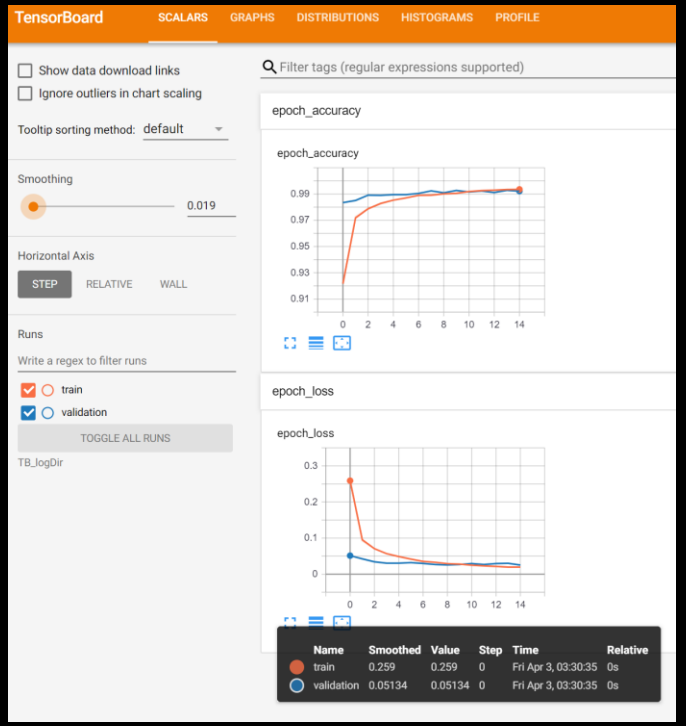

Back to our regularly scheduled programming...

TensorBoard Analysis

The most obvious thing we can do is to look at our training loss. Note that TB is happy to do this in *real-time* as the model runs. This can be very useful for you to monitor overfitting.



Our First Model
64 Wide FC



Our CNN

Early Stopping

Of course, manual monitoring is not he only way to prevent needless training after the loss has plateaued. There are numerous ways in any framework to enable early stopping.

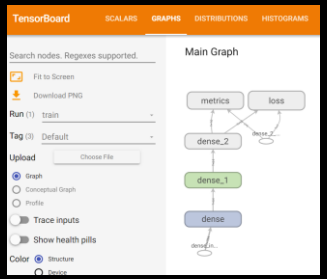
The most common is to add the *EarlyStopping* callback to your training loop.

```
tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    min_delta=0,
    patience=0,
    verbose=0,
    mode='auto',
    baseline=None,
    restore_best_weights=False,
    start_from_epoch=0
)
```

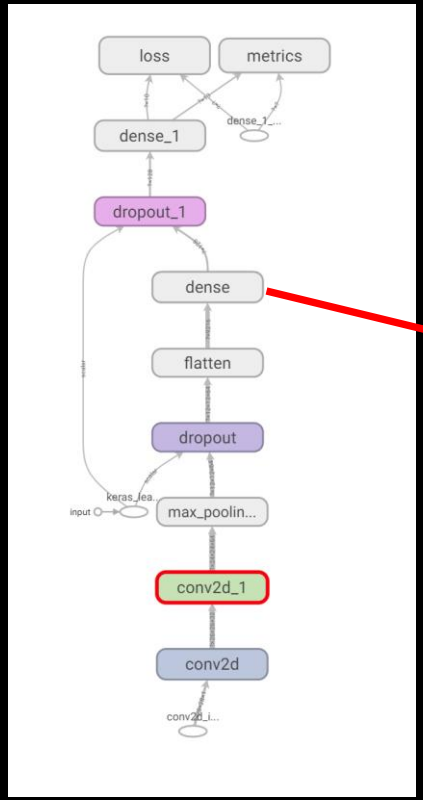
`monitor`	Quantity to be monitored. Defaults to `"val_loss"`.
`min_delta`	Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement. Defaults to `0`.
`patience`	Number of epochs with no improvement after which training will be stopped. Defaults to `0`.
`verbose`	Verbosity mode, 0 or 1. Mode 0 is silent, and mode 1 displays messages when the callback takes an action. Defaults to `0`.
`mode`	One of `{"auto", "min", "max"}`. In `min` mode, training will stop when the quantity monitored has stopped decreasing; in `"max"` mode it will stop when the quantity monitored has stopped increasing; in `"auto"` mode, the direction is automatically inferred from the name of the monitored quantity. Defaults to `"auto"`.
`baseline`	Baseline value for the monitored quantity. If not `None`, training will stop if the model doesn't show improvement over the baseline. Defaults to `None`.
`restore_best_weights`	Whether to restore model weights from the epoch with the best value of the monitored quantity. If `False`, the model weights obtained at the last step of training are used. An epoch will be restored regardless of the performance relative to the `baseline`. If no epoch improves on `baseline`, training will run for `patience` epochs and restore weights from the best epoch in that set. Defaults to `False`.
`start_from_epoch`	Number of epochs to wait before starting to monitor improvement. This allows for a warm-up period in which no improvement is expected and thus training will not be stopped. Defaults to `0`.

TensorBoard Graph Views

We can explore the architecture of the deep learning graphs we have constructed.

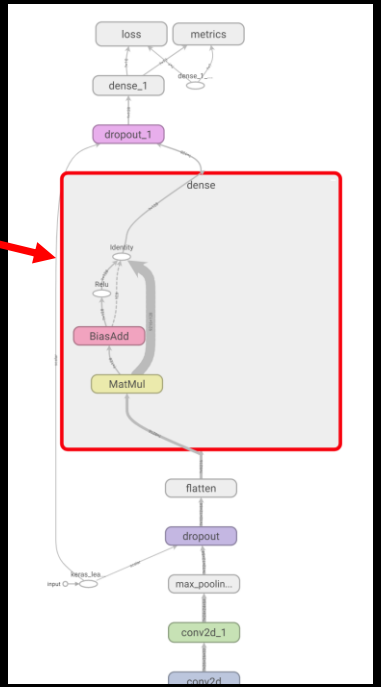


Our First Model
64 Wide FC

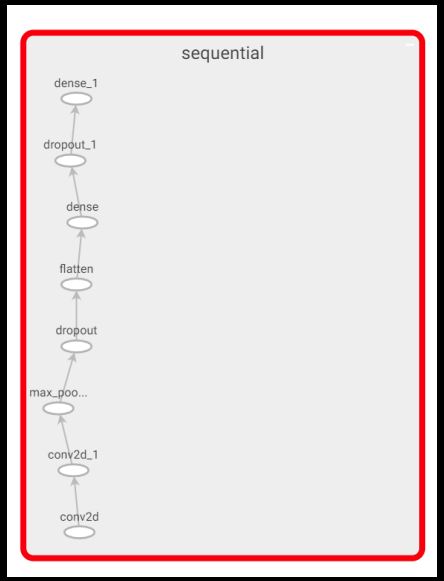


Our CNN

And we can drill down.



Our CNN's
FC Layer

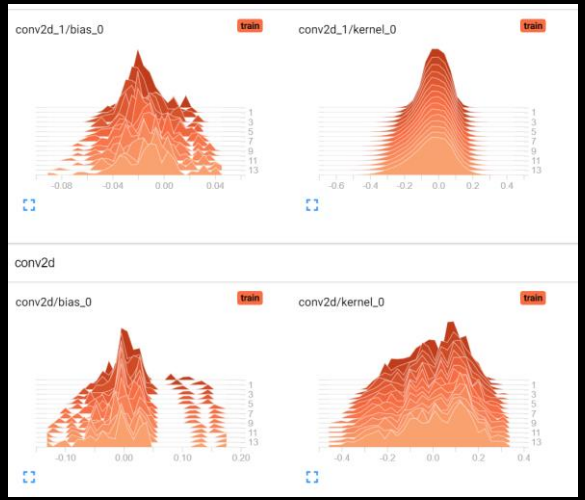


Keras
"Conceptual
Model"
View
of CNN

TensorBoard Parameter Visualization

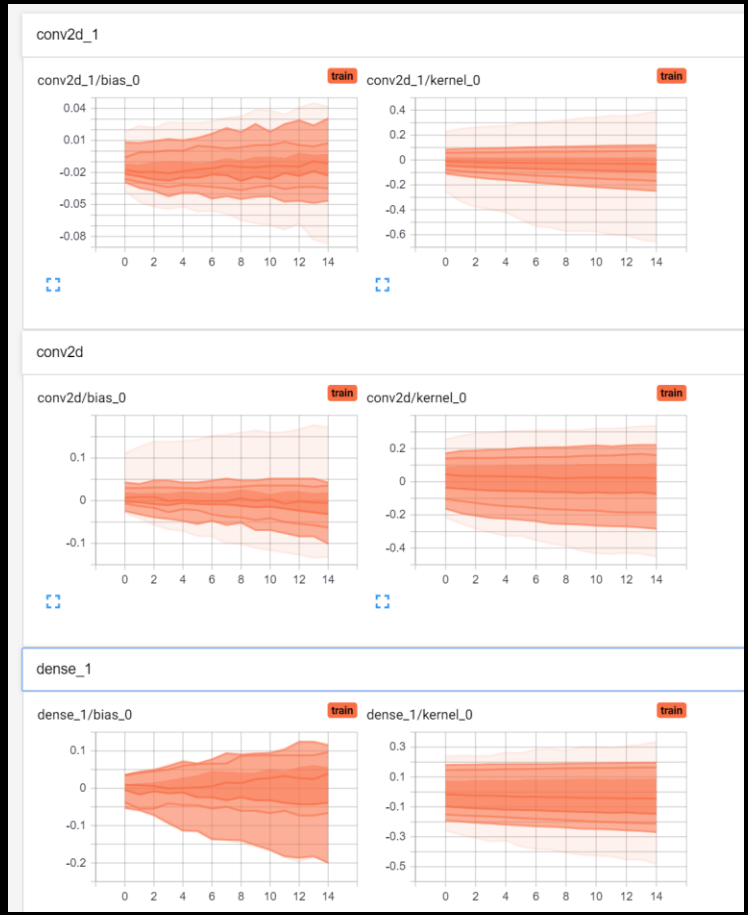
And we can observe the time evolution of our weights and biases, or at least their distributions.

This can be very telling, but requires some deeper application and architecture dependent understanding.



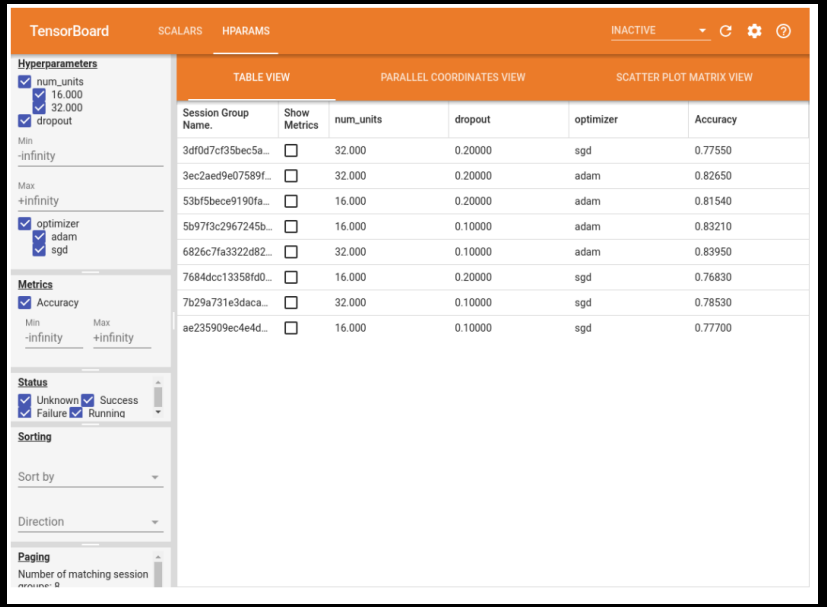
Histogram View

Distribution View



TensorBoard Add Ons

TensorBoard has lots of extended capabilities. Two particularly useful and powerful ones are Hyperparameter Search and Performance Profiling.



Hyperparameter Search

Requires some scripting on your part. Look at https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams for a good introduction.

Performance Profiling

Going beyond basics, like IO time, requires integration of hardware specific tools. This is well covered if you are using NVIDIA, otherwise you may have a little experimentation to do. The end result is a user friendly interface and valuable guidance.

Scaling

If one GPU is good, more must be better! This is largely true, and you will notice our GPU nodes are stuffed full with 4 or more GPUs each.

You might also notice that most of the machines in the "Top 10" have a lot of GPUs in them. They deliver most of the FLOPS for scientific codes, but are also an enviable Deep Learning resource.

You might have noticed that most of the interesting leading-edge research seems to involve a lot of GPUs these days.

And the very public battles in the Large Language Model space seem to be about who can get their hands on the largest GPU clusters.

How might you reach these levels of capability?

And what about those scaling limitations I mentioned earlier?

Actually a Crypto miner. We hate these guys for hoarding our GPUs!!!

Data Parallelism

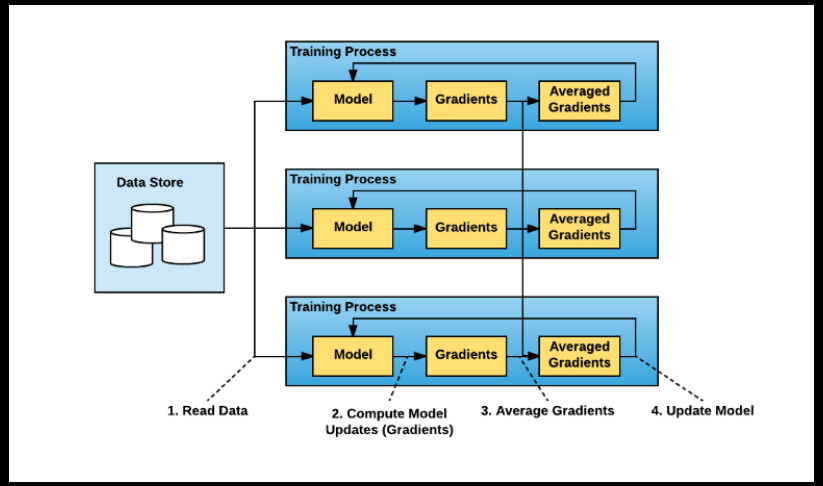
One early technique to utilize multiple GPUs was to independently train an *ensemble* of GPUs on the same task, and then have them vote on the answer. This method does work, but then the end user needs to have an ensemble of their own GPUs. This is not ideal for an application that you wish to run on your phone, or in a self-driving car.

It would be better if we could use a lot of GPUs for the training step, but end up with one great set of parameters that will fit on a single GPU when we are done. Then our users don't need to own supercomputers.

One technique to achieve this is to use *Data Parallelism* so that each GPU trains on a separate batch of data, and at the end of that batch we average the collective wisdom of all of these GPUs to arrive at our new and improved parameters.

Now when we finish we have one super set of parameters that fits on a single GPU.

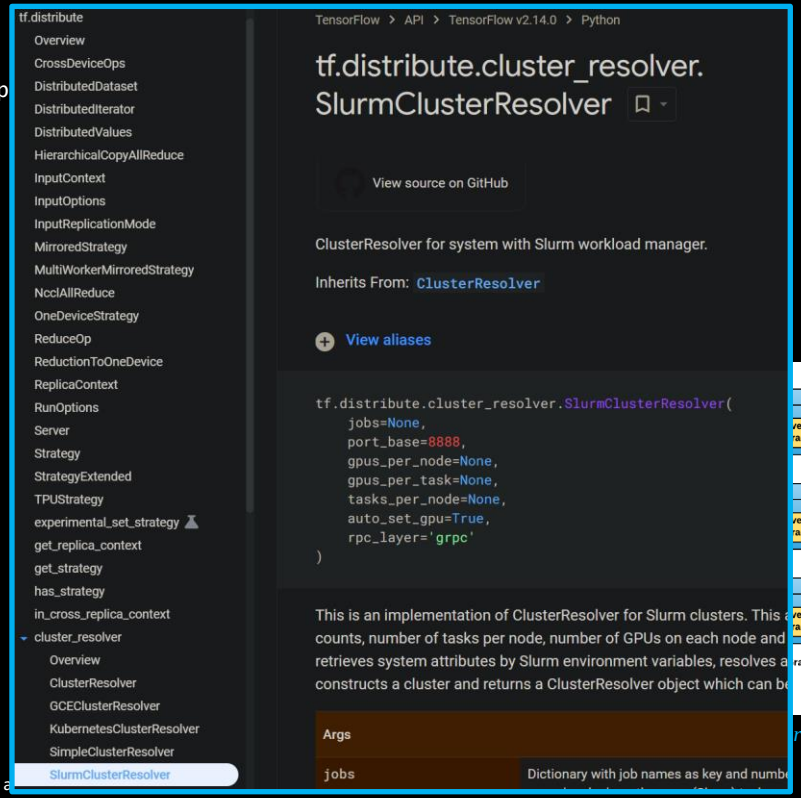
This gradient averaging requires an *all-reduce*, which can be quite expensive given the number of weights involved.



TensorFlow Scalability

This is very straightforward to implement in TensorFlow using the MirroredStrategy on a single node with multiple GPUs, or MultiWorkerMirroredStrategy across multiple nodes.

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Dropout(rate=0.2, input_shape=X.shap
        tf.keras.layers.Dense(units=64, activation='relu'),
        ...
    ])
    model.compile(...)
model.fit(...)
```



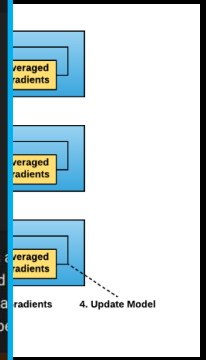
An alternative that has proven itself at extreme scale is *Horovod*.

MNIST with Horovod!

```
# Horovod: initialize Horovod.
hvd.init()

# Horovod: pin GPU to be used to process local rank (one GPU per process)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = str(hvd.local_rank())
K.set_session(tf.Session(config=config))
...
# Horovod: adjust number of epochs based on number of GPUs.
epochs = int(math.ceil(12.0 / hvd.size()))
...
# Horovod: adjust learning rate based on number of GPUs.
opt = keras.optimizers.Adadelta(1.0 * hvd.size())
...
# Horovod: add Horovod Distributed Optimizer.
opt = hvd.DistributedOptimizer(opt)
...
model.compile(loss=keras.losses.categorical_crossentropy,optimizer=opt,metrics=['a

callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(0),]
if hvd.rank() == 0: callbacks.append(keras.callbacks.ModelCheckpoint('./checkpoint-
{epoch}.h5'))
```

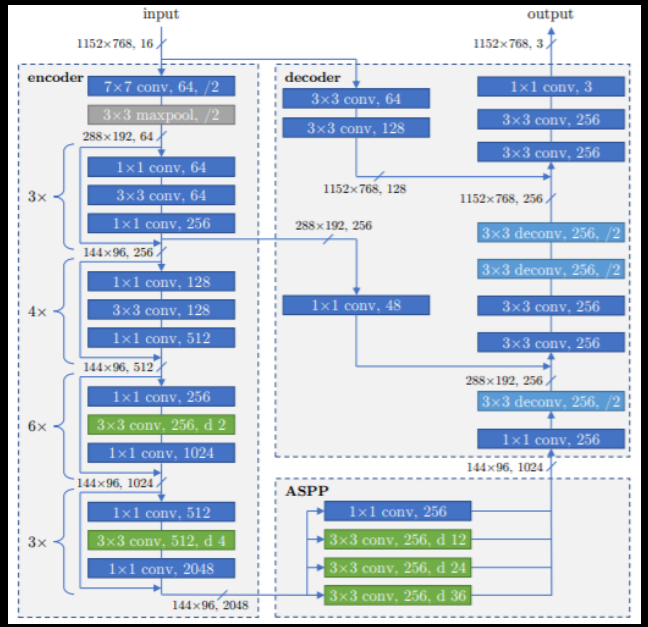


rning in TensorFlow

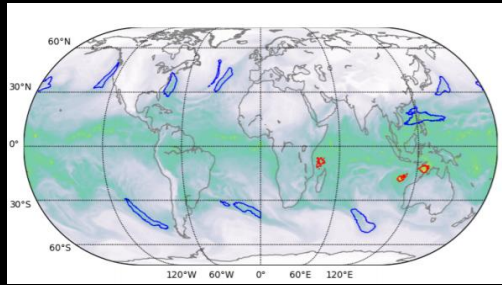
You can find a full example of using Horovod with a Keras MNIST code at:
https://horovod.readthedocs.io/en/latest/keras.html

Scaling Up Massively

Horovod demonstrates its excellent scalability with a Climate Analytics code that won the Gordon Bell prize in 2018. It predicts Tropical Cyclones and Atmospheric River events based upon climate models. It shows not only the reach of deep learning in the sciences, but the scale at which networks can be trained.



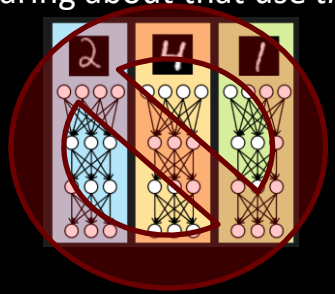
- *1.13 ExaFlops (mixed precision) peak training performance*
- *On 4560 6 GPU nodes (27,360 GPUs total)*
- *High-accuracy (harder when predicting "no hurricane today" is 98% accurate), solved with weighted loss function.*
- *Layers each have different learning rate*



Exascale Deep Learning for Climate Analytics
Kurth, et. al.

Model Parallelism

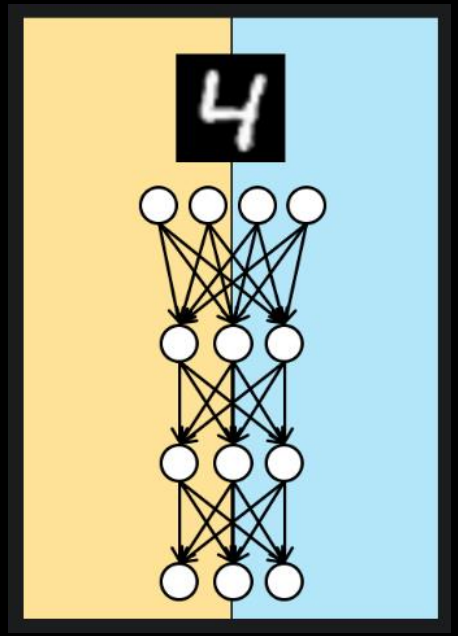
What about all these LLMs you have been hearing about that use *trillions* of parameters? Now, we don't have enough memory to fit the whole model on one GPU.



Instead we spread the parts of the model around (mostly their parameters, but could also be different sections of a more complex model) by using *Model Parallelism.*

The most popular way to do this in TensorFlow is via the *Mesh TensorFlow* API.

And, we can mix the way we distribute these parameters, layers, pipelines and model branches in various hybrid methods as well.



From the Chainer *docs on their parallelism API. Yet another DL framework.*

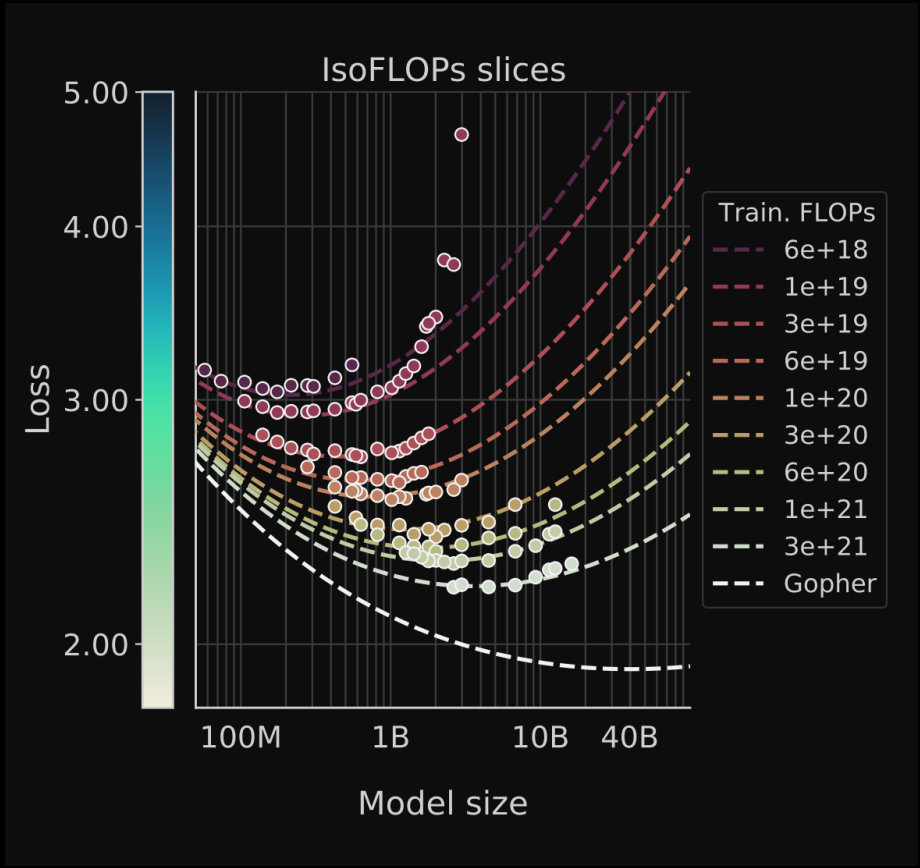
Scaling of LLMs

For LLMs that have been designed to scale well (avoiding overfitting and vanishing gradients, for example), we find that performance is a predictable function of:

- The dataset size
- The number of parameters

And these curves show no signs of ending yet. Or do they?

So, in these applications we do expect better performance through brute force scaling.



Training Compute-Optimal Large Language Models
Hoffmann et. al.

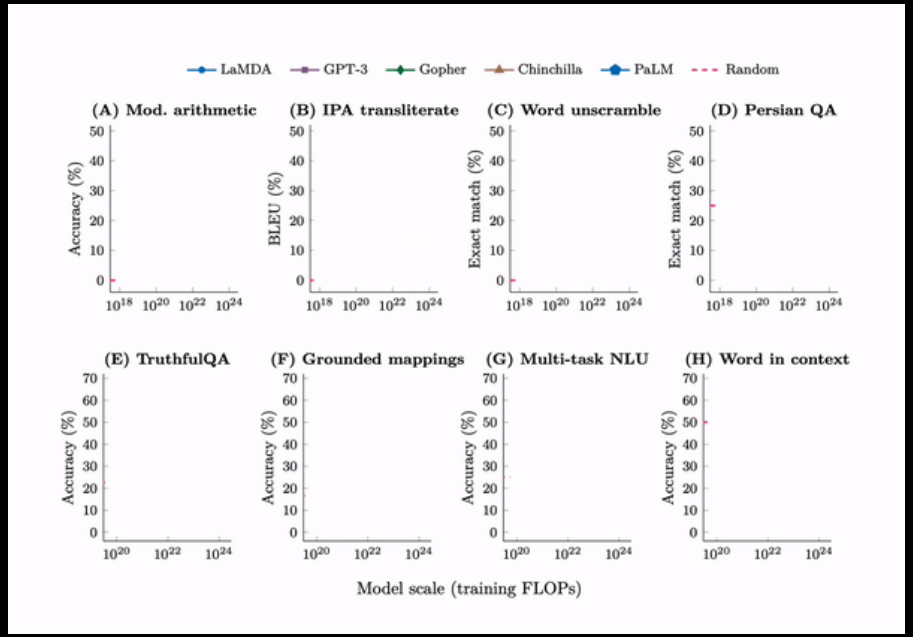
Emergent Capability

An emergent ability as an ability that is not present in small models but develops as the model is scaled. These might be unanticipated.

Some areas of science are familiar with this idea (physics and biology, for sure). In computer science this concept has largely been a novelty (Conway's Game of Life is a notable example). In Deep Learning, it has become a very significant phenomena.

Once again, bear in mind that many of the principles we have mentioned (overfitting, vanishing gradients, etc.) mean that brute force scaling is not going to be a default route to better performance.

Instead, understanding of those principles will allow you the option to scale.

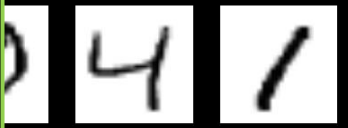


https://www.jasonwei.net/blog/emergence
Also a good associated paper.

Data Augmentation

As I've mentioned, labeled data is valuable. This type of *supervised learning* often requires human-labeled data. Getting more out of our expensive data is very desirable. More datapoints generally equals better accuracy. The process of generating more training data from our existing pool is called *Data Augmentation*, and is an extremally common technique, especially for classification

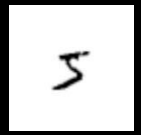
Our MNIST network has learned to recognize ve



What if we wanted to teach it:

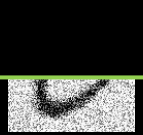
How many samples do we need?

This is another hyperparameter (yes), where we can only offer a vague rule of thumb. And that suggestion is about 5000 per category for competence, 10 million for a real task with human performance.



Scale Invariance

Rotation Invariance



Noise Tolerance

Translation Invariance

You can see how straightforward and mechanical this is. And yet very effective. You will often see detailed explanations of the data augmentation techniques employed in any given project.

Note that tf.image makes many of these processes very convenient.

Stupid Neural Nets

Why can't they learn like we do? Can't I just tell you a fact, or an algorithm, and you can just "get it" it without countless iteration?

English for Infants

Idiots Guide to Winning the US Open

One-Shot Learning

On the other hand, maybe I could teach adult you what a platypus is with one example. And, if you want to spot a particular mad bomber with your airport facial recognition system, you may only have one photo.

There is such a thing as "one-shot" (or N-shot) learning. But it is harder, requires more specialized techniques, and is straying into the area of unsupervised learning. We will come back to this, but it is no magic bullet for sparse data.

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

```
def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
```

```
n'.format(
```

```
pose([
```

PyTorch CNN MNIST

Not a fair comparison of terseness as this version has a lot of extra flexibility.

From:

https://github.com/pytorch/examples/blob/master/mnist/main.py

```
    model = Net().to(device)
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        test(args, model, device, test_loader)
        scheduler.step()

    if args.save_model:
        torch.save(model.state_dict(), "mnist_cnn.pt")


if __name__ == '__main__':
    main()
```

More tf.keras.datasets Fun

CRIM	per capita crime rate by town
ZN	proportion of residential land
INDUS	proportion of non-retail busine
CHAS	Charles River dummy variable (=
NOX	nitric oxides concentration (pa
RM	average number of rooms per dwe
AGE	proportion of owner-occupied un
DIS	weighted distances to five Bost
RAD	index of accessibility to radia
TAX	full-value property-tax rate pe
PTRATIO	pupil-teacher ratio by town
B	1000(Bk - 0.63)^2 where Bk is t
LSTAT	% lower status of the populatio
MEDV	Median value of owner-occupied

Boston Housing — Predict housing prices base upon crime, zoning, pollution, etc.

CIFAR10 — 32x32 color images in 10 classes.

CIFAR100 — Like CIFAR10 but with 100 non-overlapping classes.

IMDB — 1 sentence positive or negative reviews.

> *I have been known to fall asleep during films, but this...*
> Mann photographs the Alberta Rocky Mountains in a superb fashion...
> This is the kind of film for a snowy Sunday afternoon...

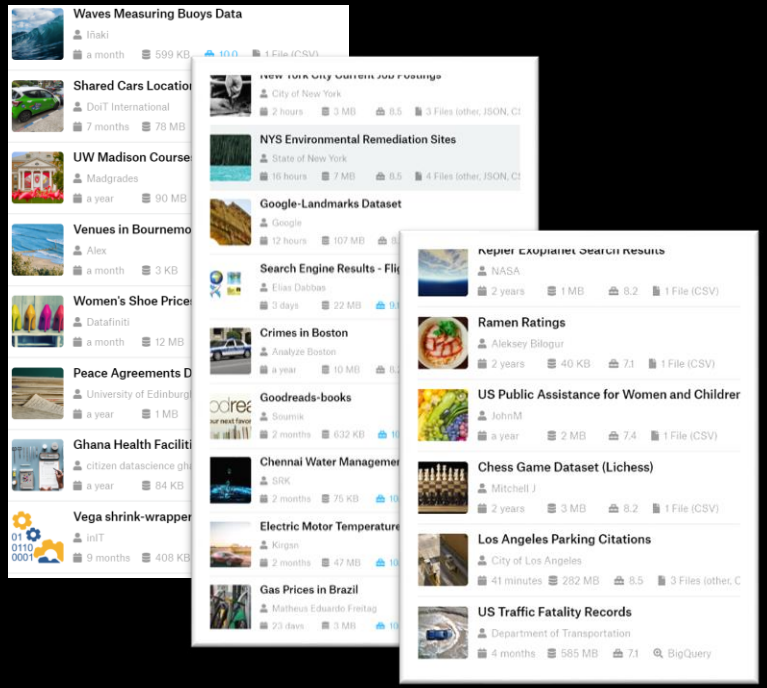
Reuters — 46 topics in newswire form.

> Its december acquisition of space co it expects earnings per share in 1987 of 1 15 to 1 30 dlrs per share up from 70 cts in 1986 the company said pretax net should rise to nine to 10 mln dlrs from six mln dlrs in 1986 and rental operation revenues to 19 to 22 mln dlrs from 12 5 mln dlrs it said cash flow per share this year should be 2 50 to three dlrs reuters...

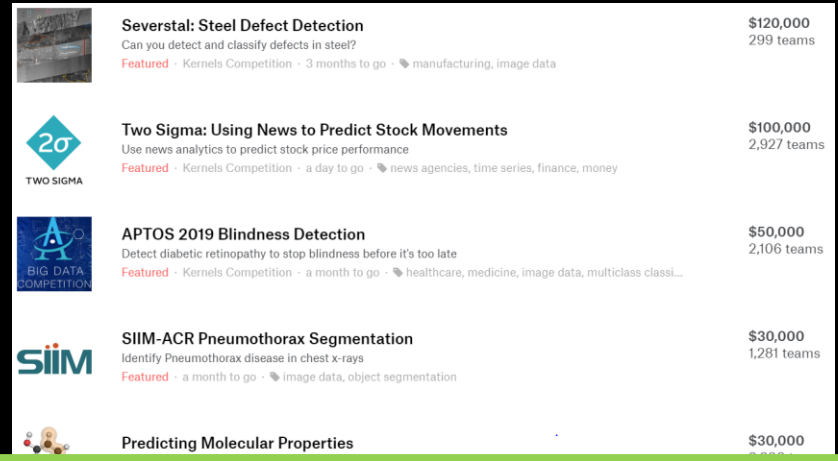
Endless Exercises

Kaggle Challenge

The benchmark driven nature of deep learning research, and its competitive consequences, have found a nexus at Kaggle.com. There you can find over 20,000 datasets:



and competitions:



Including this one: