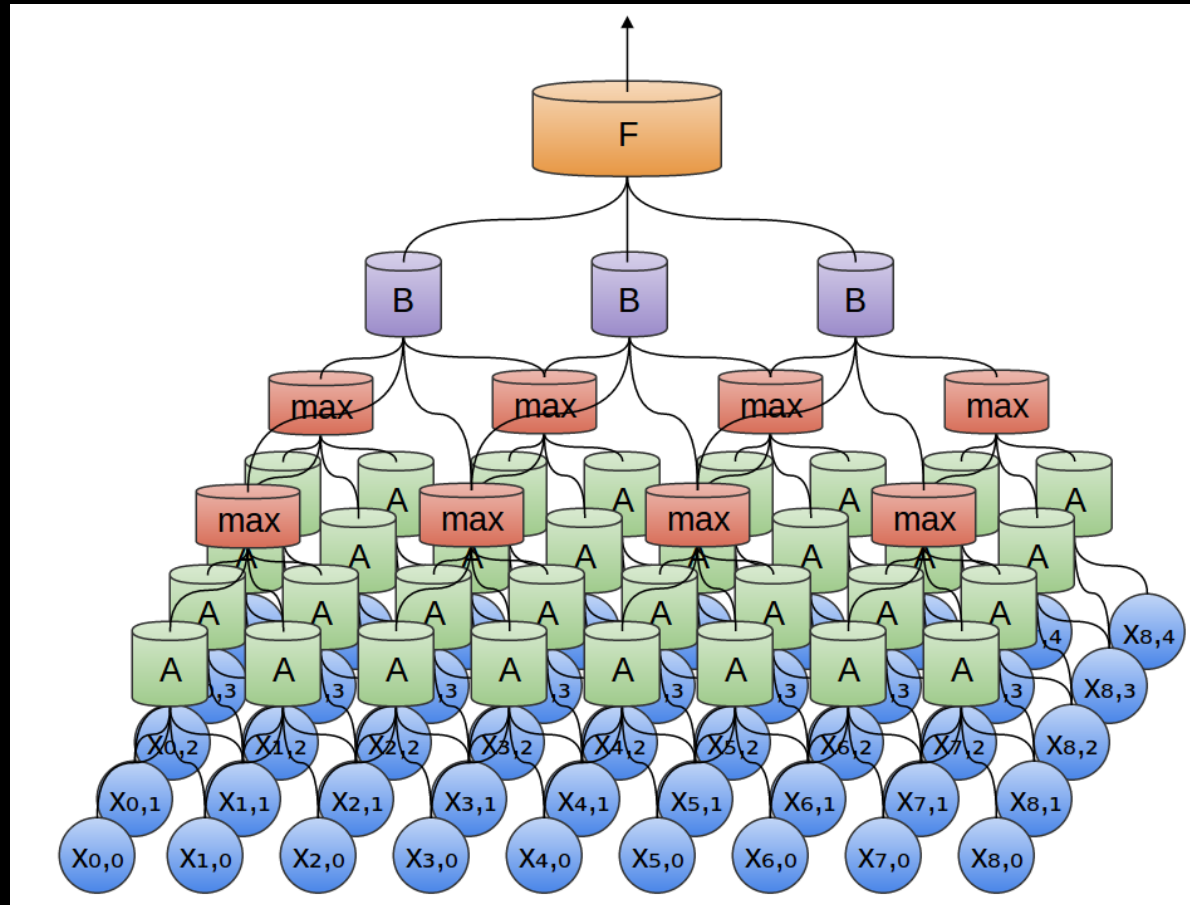
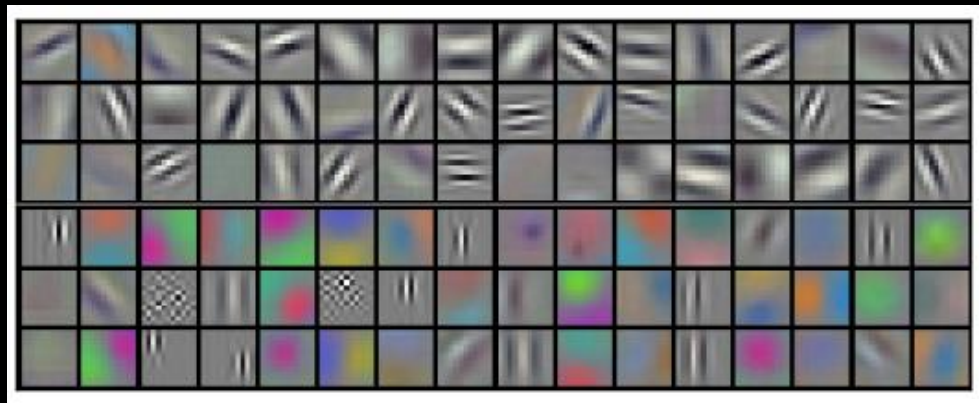


Pooling

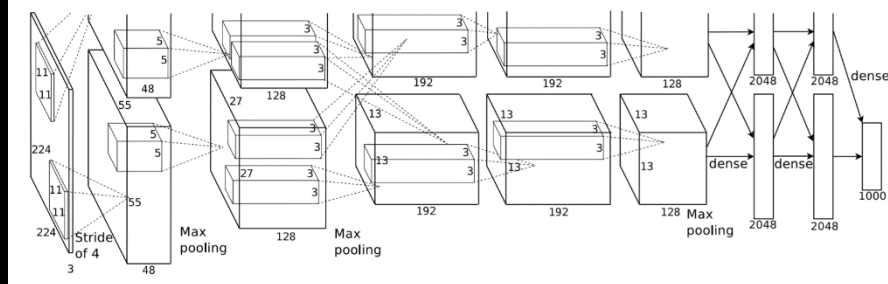


A Groundbreaking Example

These are the 96 first layer 11x11 (x3, RGB, stacked here) filters from AlexNet.



Among the several novel techniques combined in this work (such as aggressive use of ReLU), they used dual GPUs, with different flows for each, communicating only at certain layers. A result is that the bottom GPU consistently specialized on color information, and the top did not.



Let's Start Small

```
import tensorflow as tf

mnist = tf.keras.datasets.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.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation='relu'),
    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))
```

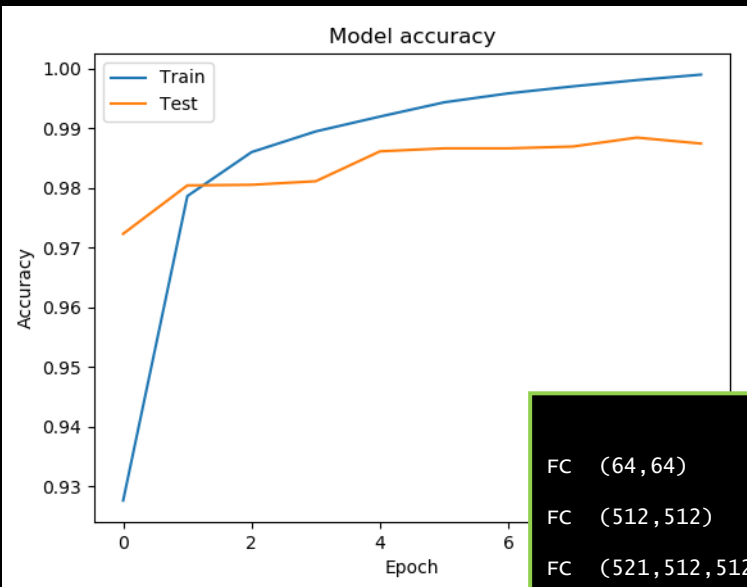
Early CNN Results

....

....

Epoch 10/10

60000/60000 [=====] - 12s 198us/sample - loss: 0.0051 - accuracy: 0.9989 - val_loss: 0.0424 - val_accuracy: 0.9874



Score Thus Far

FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7

Primitive CNN

model.summary()

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1	(None, 13, 13, 32)	0
flatten_1 (Flatten)	(None, 5408)	0
dense_38 (Dense)	(None, 100)	540900
dense_39 (Dense)	(None, 10)	1010
=====		

ms: 542,230
params: 542,230
ble params: 0

Scaling Up The CNN

```
import tensorflow as tf

mnist = tf.keras.datasets.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.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

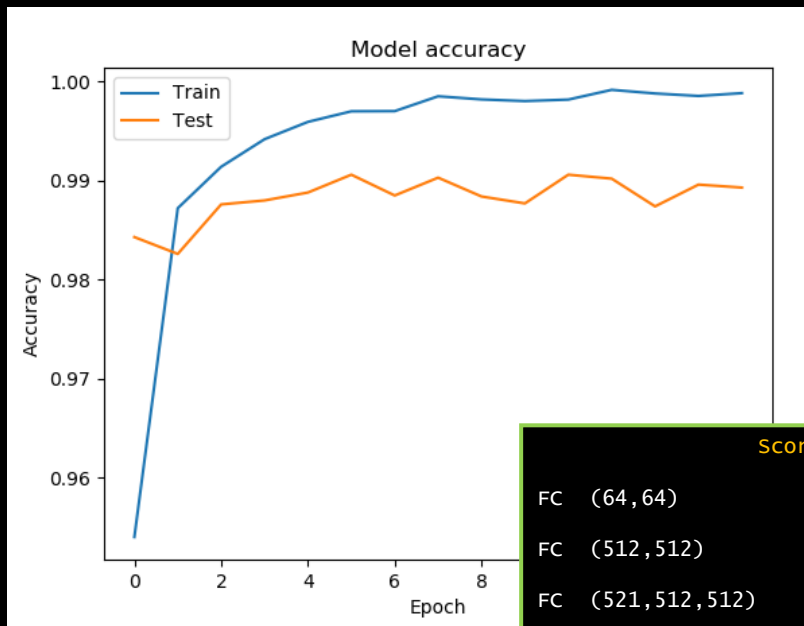
model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

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

Deeper CNN Results

....
....

Epoch 15/15
60000/60000 [=====] - 34s 566us/sample - loss: 0.0052 - accuracy: 0.9985 - val_loss: 0.0342 - val_accuracy: 0.9903



Score Thus Far

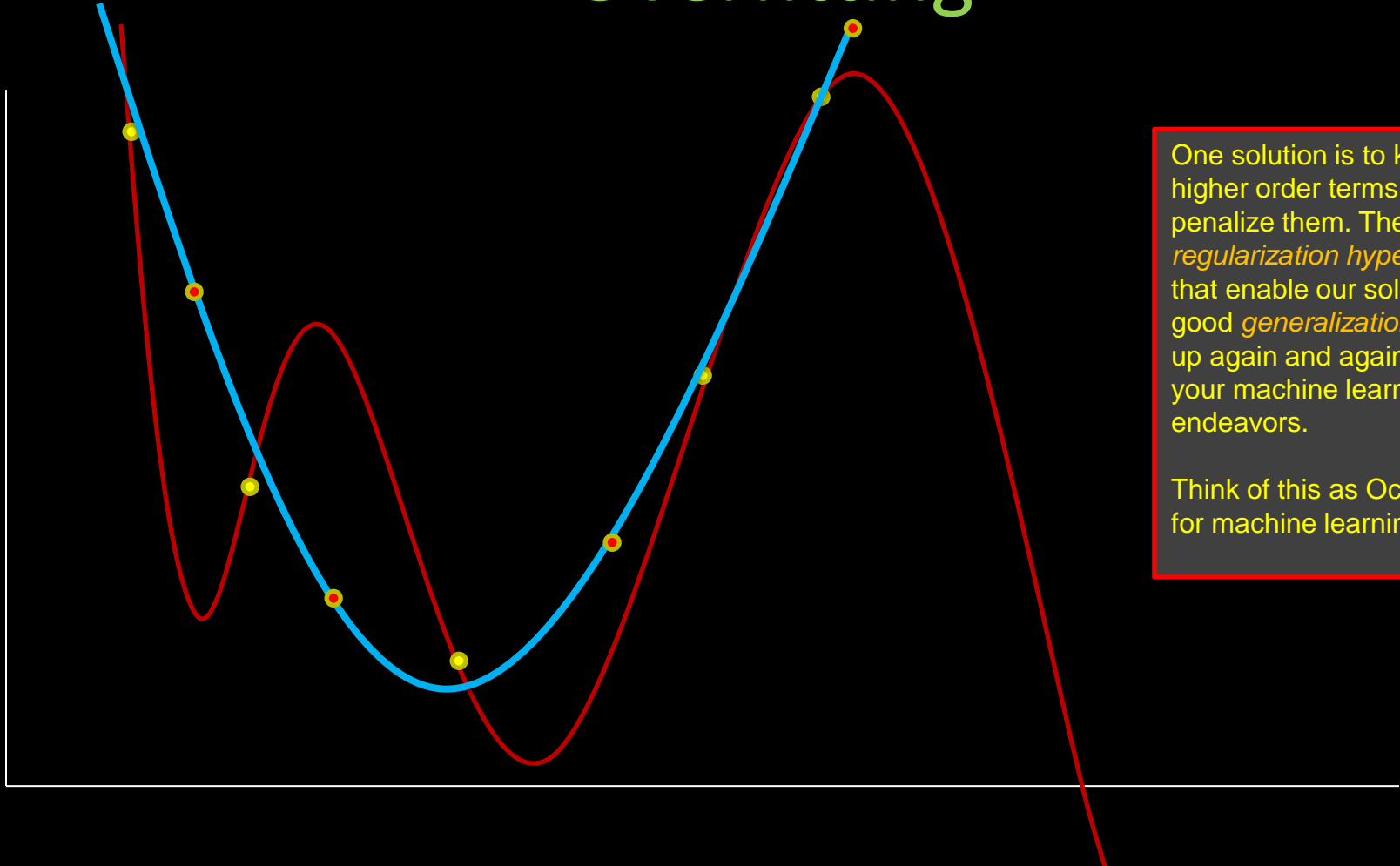
FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7
CNN (2 Layer)	99.0

Deeper CNN

```
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
conv2d_4 (Conv2D)	(None, 26, 26, 32)	320
conv2d_5 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_3	(None, 12, 12, 64)	0
flatten_3 (Flatten)	(None, 9216)	0
dense_42 (Dense)	(None, 128)	1179776
	(None, 10)	1290
=====		
Total params: 1,199,882		
Trainable params: 0		

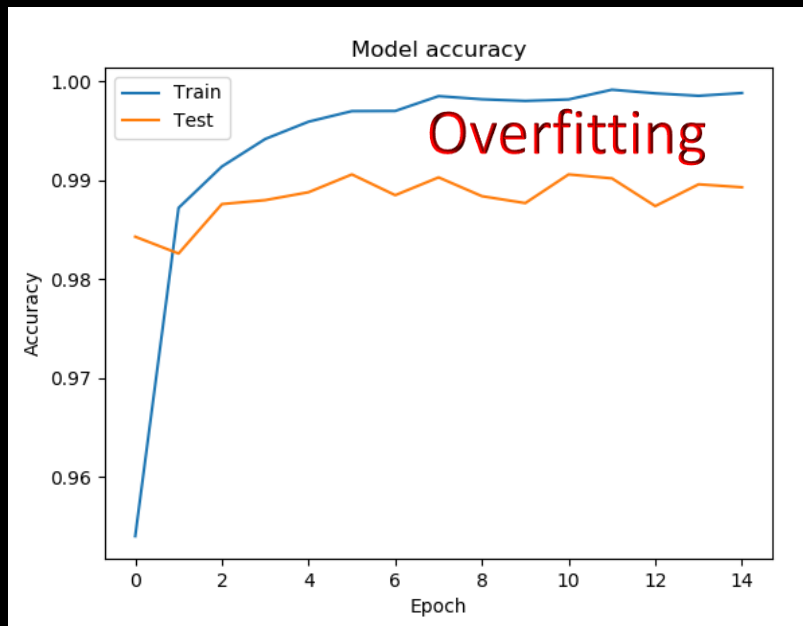
Overfitting



One solution is to keep using higher order terms, but to penalize them. These *regularization hyperparameters* that enable our solution to have good *generalization* will show up again and again throughout your machine learning endeavors.

Think of this as Occam's Razor for machine learning.

Dropout



We need some form of regularization to help with the overfitting. One seemingly crazy way to do this is the relatively new technique (introduced by the venerable Geoffrey Hinton in 2012) of Dropout.



Some view it as an ensemble method that trains multiple data models simultaneously. One neat perspective of this analysis-defying technique comes from Jürgen Schmidhuber, another innovator in the field; under certain circumstances, it could also be viewed as a form of training set augmentation: effectively, more and more informative complex features are removed from the training data.

CNN With Dropout

```
import tensorflow as tf

mnist = tf.keras.datasets.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))
```

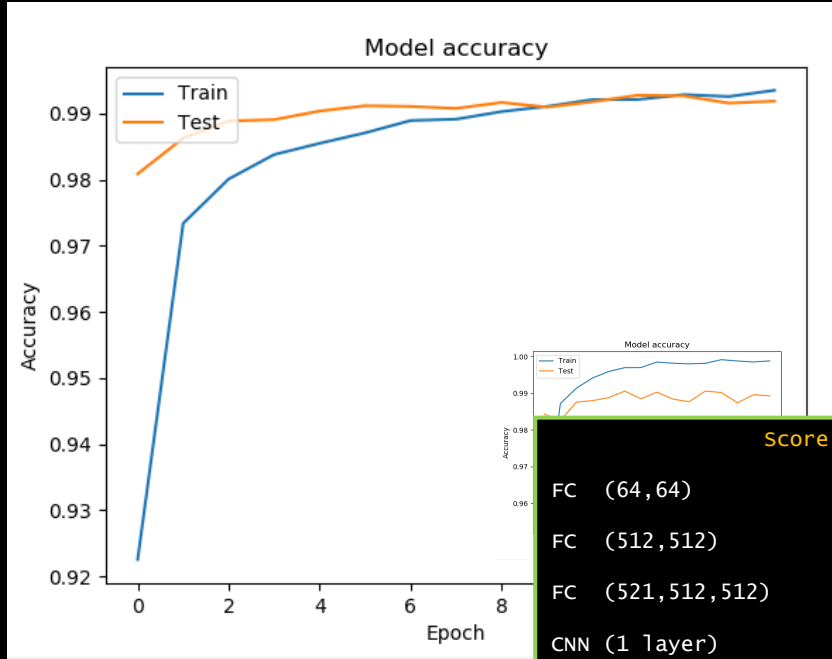
Parameter is fraction to *drop*.

Help From Dropout

....
....

Epoch 15/15

60000/60000 [=====] - 40s 667us/sample - loss: 0.0187 - accuracy: 0.9935 - val_loss: 0.0301 - val_accuracy: 0.9919



Score Thus Far

FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7
CNN (2 Layer)	99.0
CNN with Dropout	99.2

Dropout CNN

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 26, 26, 32)	320
conv2d_13 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_7	(None, 12, 12, 64)	0
dropout_4 (Dropout)	(None, 12, 12, 64)	0
flatten_7 (Flatten)	(None, 9216)	0
(Dense)	(None, 128)	1179776
(Dropout)	(None, 128)	0
(Dense)	(None, 10)	1290

params: 1,199,882
trainable params: 0

Batch Normalization

Another "between layers" layer that is quite popular is Batch Normalization. This technique really helps with vanishing or exploding gradients. So it is better with deeper networks.

- Maybe not so compatible with Dropout, but the subject of research (and debate).
- Maybe Apply Dropout after all BN layers: <https://arxiv.org/pdf/1801.05134.pdf>
- Before or after non-linear activation function? Oddly, also open to debate. But, it may be more appropriate after the activation function if for s-shaped functions like the hyperbolic tangent and logistic function, and before the activation function for activations that result in non-Gaussian distributions like ReLU.

How could we apply it before or after our activation function if we wanted to? We haven't been peeling our layers apart, but we can micro-manage more if we want to:

```
model.add(tf.keras.layers.Conv2D(64, (3, 3), use_bias=False))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation("relu"))

model.add(tf.keras.layers.Conv2D(64, kernel_size=3, strides=2, padding="same"))
model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
model.add(tf.keras.layers.BatchNormalization(momentum=0.8))
```

There are also normalizations that work on single samples instead of batches, so better for recurrent networks. In TensorFlow we have Group Normalization, Instance Normalization and Layer Normalization.

Trying Batch Normalization

```
import tensorflow as tf

mnist = tf.keras.datasets.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.BatchNormalization(),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation='softmax')
])

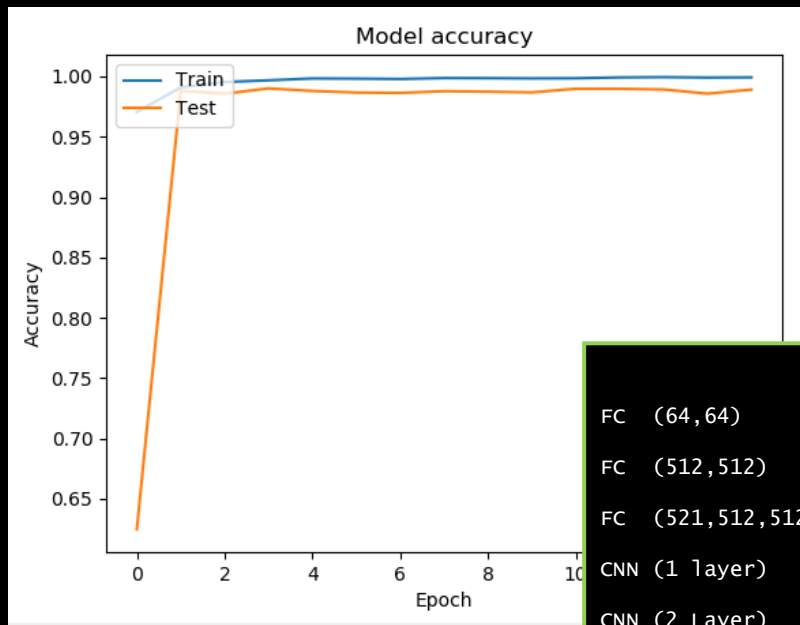
model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

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

Not So Helpful

....

Epoch 15/15
60000/60000 [=====] - 50s 834us/sample - loss: 0.0027 - accuracy: 0.9993 - val_loss: 0.0385 - val_accuracy: 0.9891



Score Thus Far

FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7
CNN (2 Layer)	99.0
CNN with Dropout	99.2
Batch Normalization	98.9

Batch Normalization CNN

model.summary()

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
batch_normalization	(None, 26, 26, 32)	128
conv2d_3 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1	(None, 12, 12, 64)	0
batch_normalization_1	(None, 12, 12, 64)	256
flatten	(None, 9216)	0
conv2d_4 (Conv2D)	(None, 128)	1179776
batch_normalization_2	(Batch Normalization)	512
conv2d_5 (Conv2D)	(None, 10)	1290
Total params: 1,200,330		
Trainable params: 1,200,330		
Non-trainable params: 448		

Real Time Demo

This *amazing, stunning, beautiful* demo from Adam Harley (now just across campus) is very similar to what we just did, but different enough to be interesting.

<http://scs.ryerson.ca/~aharley/vis/conv/flat.html>

It is worth experiment with. Note that this is an excellent demonstration of how efficient the forward network is. You are getting very real-time analysis from a lightweight web program. Training it took some time.

Draw your number here

2



Downsampled drawing: 2

First guess: 2

Second guess: 0

Layer visibility

Input layer ☐ Show

Convolution layer 1 ☐ Show

Downsampling layer 1 ☐ Show

Convolution layer 2 ☐ Show

Downsampling layer 2 ☐ Show

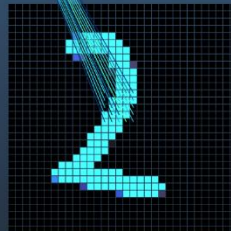
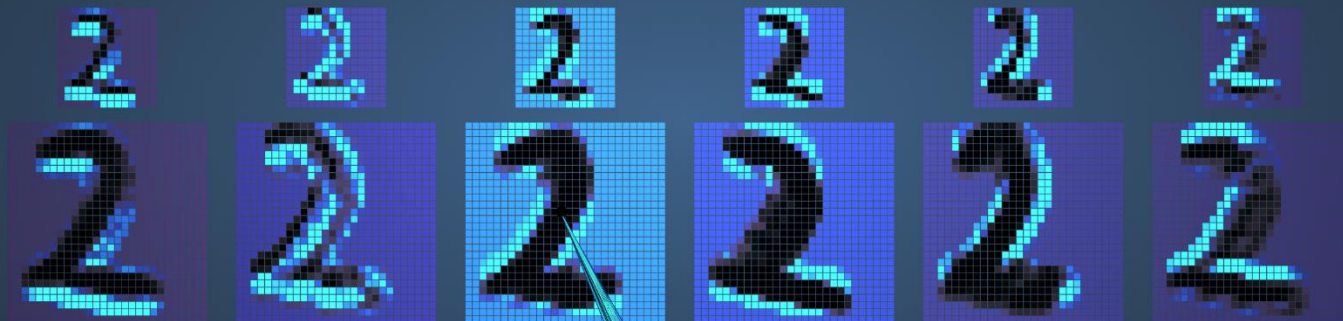
Fully-connected layer 1 ☐ Show

Fully-connected layer 2 ☐ Show

Output layer ☐ Show

0123456789

0 1 2 3 4 5 6 7 8 9



Adding TensorBoard To Your Code

TensorBoard is a very versatile tool that allows us multiple types of insight into our TensorFlow codes. We need only add a callback into the model to activate the necessary logging.

```
...  
...  
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])  
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir='TB_logDir', histogram_freq=1)  
history = model.fit(train_images, train_labels, batch_size=128, epochs=15, verbose=1,  
                    validation_data=(test_images, test_labels), callbacks=[tensorboard_callback])  
...  
...
```

TensorBoard runs as a server, because it has useful run-time capabilities, and requires you to start it separately, and to access it via a browser.

Somewhere else:

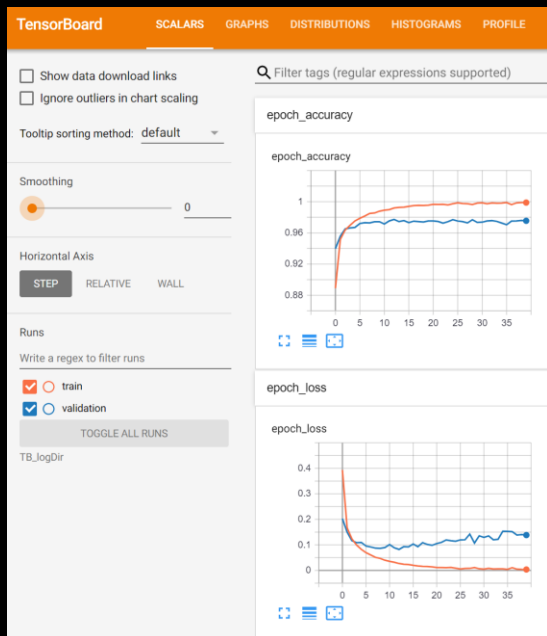
```
tensorboard --logdir=TB_logD
```

Somewhere else:

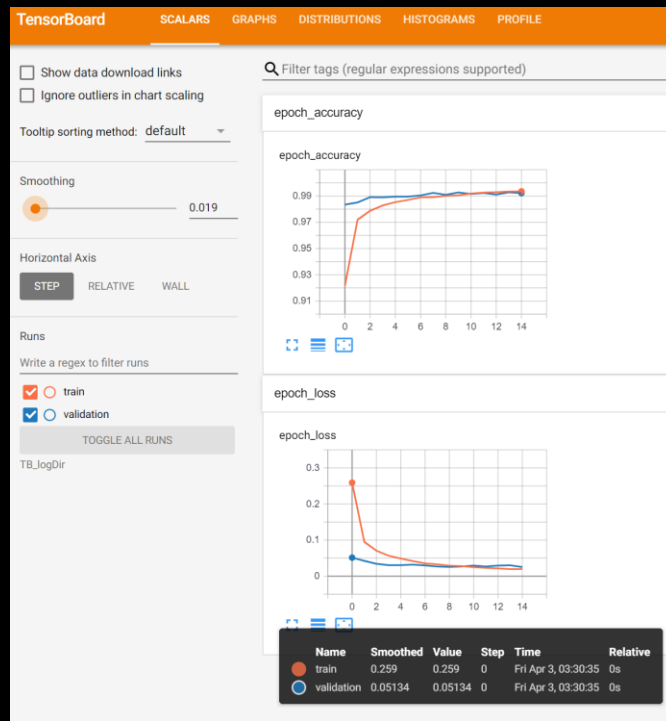
```
Start your Browser and point it at port 6006: http://localhost:6006/
```


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.



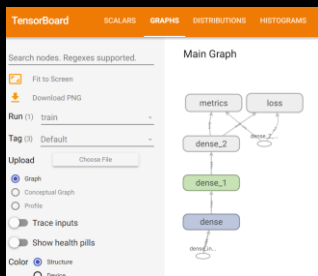
Our First Model
64 Wide FC



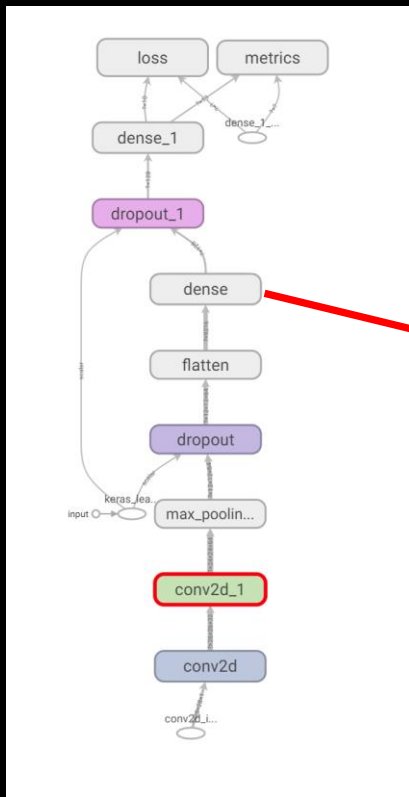
Our CNN

TensorBoard Graph Views

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

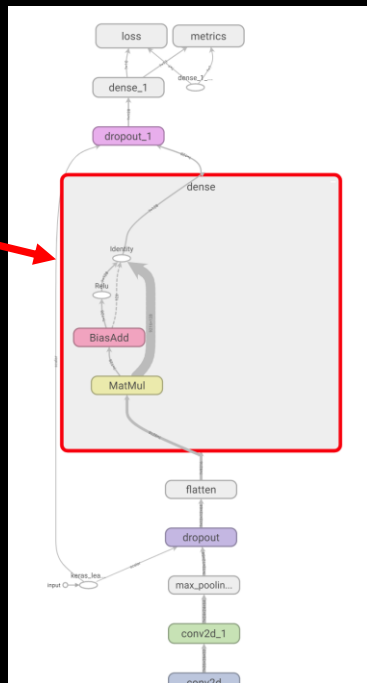


Our First Model
64 Wide FC

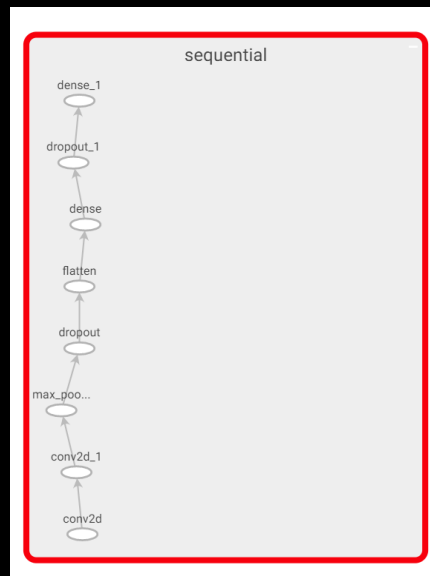


Our CNN

And we can drill down.



Our CNN's
FC Layer



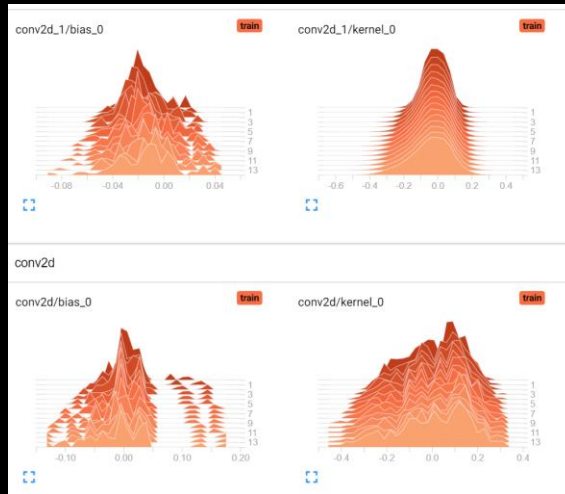
Keras
"Conceptual
Model"
View
of CNN

TensorBoard Parameter Visualization

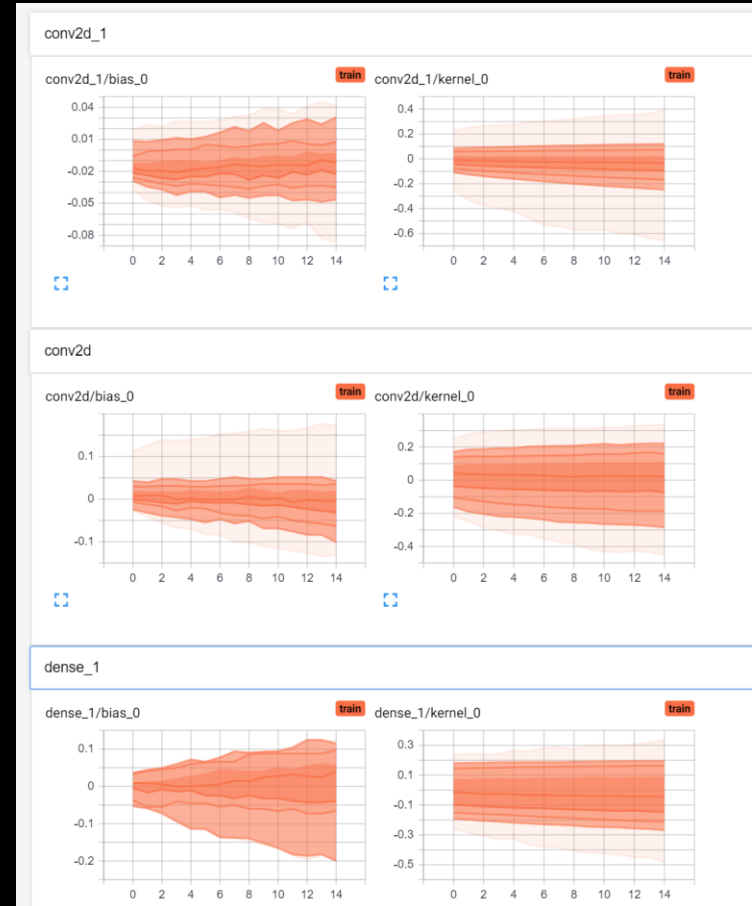
Distribution View

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.



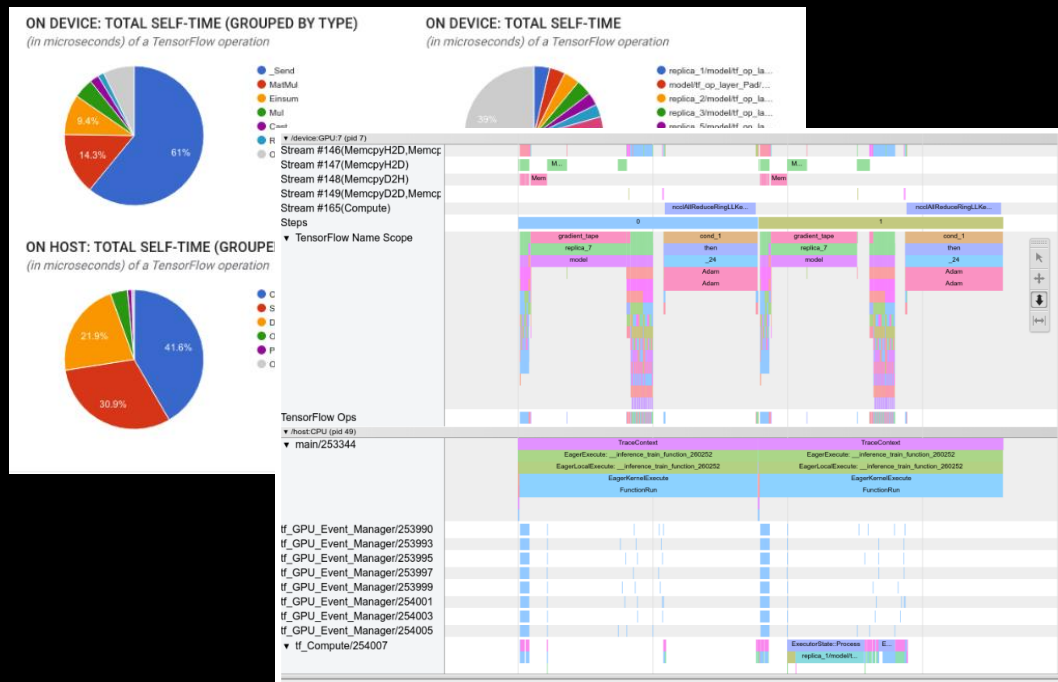
Histogram View



TensorBoard Add Ons

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

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.

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 Up

You may have the idea that deep learning has a voracious appetite for GPU cycles. That is absolutely the case, and the leading edge of research is currently limited by available resources. Researchers routinely use many GPUs to train a model. Conversely, the largest resources demand that you use them in a parallel fashion. There are capabilities built into TensorFlow, the **MirroredStrategy**.

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Dropout(rate=0.2, input_shape=x.shape[1:]),
        tf.keras.layers.Dense(units=64, activation='relu'),
        ...
    ])
    model.compile(...)
    model.fit(...)
```

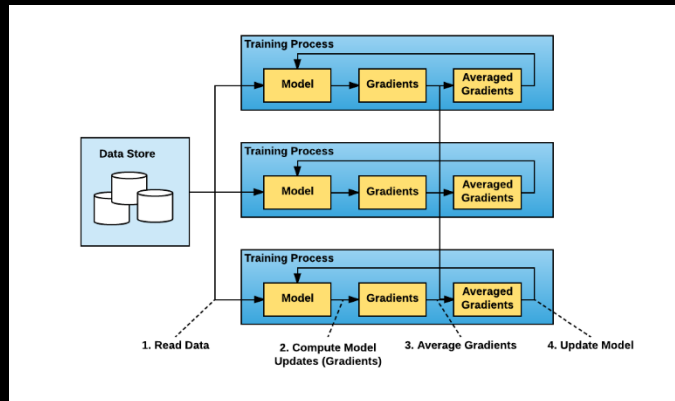
An alternative that has proven itself at extreme scale is *Horovod*.

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=['accuracy'])

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

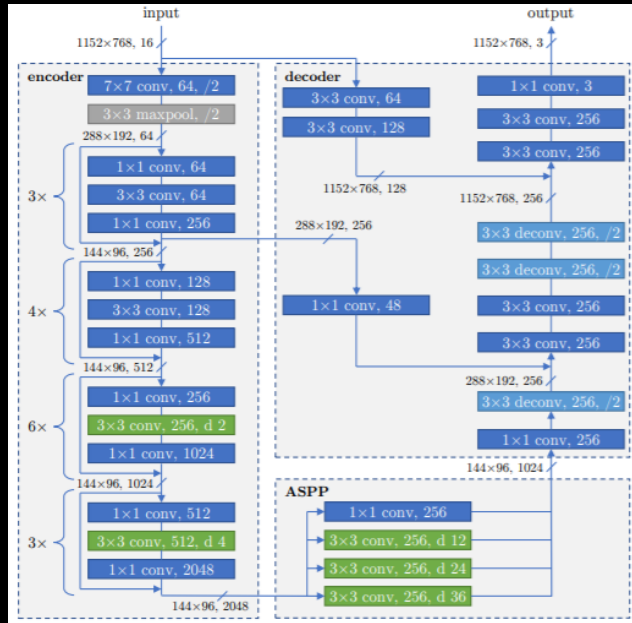


Horovod: fast and easy distributed deep learning in TensorFlow
Alexander Sergeev, Mike Del Balso

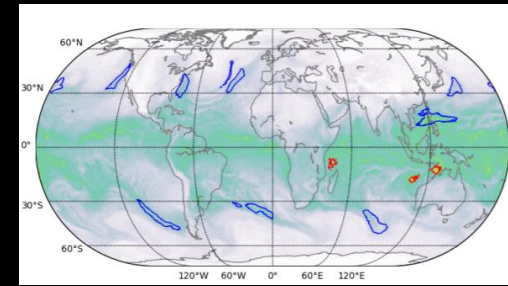
You can find a full example of using Horovod with a Keras MNIST code at <https://horovod.readthedocs.io/en/latest/keras.html>

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*



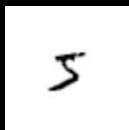
Data Augmentation

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 extremely common technique, especially for classification problems.

Our MNIST network has learned to recognize very uniformly formatted characters:



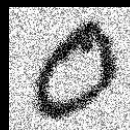
What if we wanted to teach it:



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.

```

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)

    print('Test set: Average loss: {:.4f}, Accuracy: {}/{} = {:.3f}%'.format(
        test_loss, correct, len(test_loader.dataset), 100. * correct / len(test_loader.dataset)))

if __name__ == '__main__':
    main()

```

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>

Exercises

We are going to leave you with a few substantial problems that you are now equipped to tackle. Feel free to use your extended workshop access to work on these, and remember that additional time is an easy Startup Allocation away. Of course everything we have done is standard and you can work on these problems in any reasonable environment.

You may have wondered what else was to be found at [tf.keras.datasets](https://tf.keras.datasets.fashion). The answer is many interesting problems. The obvious follow-on is:

Fashion MNIST

These are 60,000 training images, and 10,000 test images of 10 types of clothing, in 28x28 greyscale. Sound familiar? A more challenging drop-in for MNIST.



More tf.keras.datasets Fun

Boston Housing

Predict housing prices base upon crime, zoning, pollution, etc.

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 ur
DIS	weighted distances to five Bost
RAD	index of accessibility to radiat
TAX	full-value property-tax rate pe
PTRATIO	pupil-teacher ratio by town
B	$1000(B_k - 0.63)^2$ where B_k is t
LSTAT	% lower status of the populatio
MEDV	Median value of owner-occupied

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

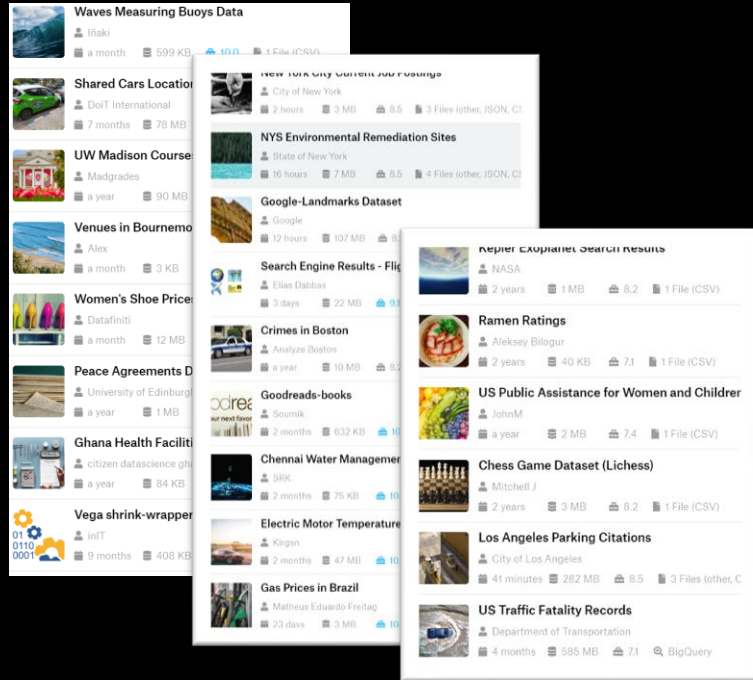
46 topics in newswire form.

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:

	Severstal: Steel Defect Detection Can you detect and classify defects in steel? <i>Featured</i> · Kernels Competition · 3 months to go · manufacturing, image data	\$120,000 299 teams
	Two Sigma: Using News to Predict Stock Movements Use news analytics to predict stock price performance <i>Featured</i> · Kernels Competition · a day to go · news agencies, time series, finance, money	\$100,000 2,927 teams
	APTOS 2019 Blindness Detection Detect diabetic retinopathy to stop blindness before it's too late <i>Featured</i> · Kernels Competition · a month to go · healthcare, medicine, image data, multiclass classi...	\$50,000 2,106 teams
	SIIM-ACR Pneumothorax Segmentation Identify Pneumothorax disease in chest x-rays <i>Featured</i> · a month to go · image data, object segmentation	\$30,000 1,281 teams
	Predicting Molecular Properties	\$30,000 0 teams

Including this one:

	Digit Recognizer Learn computer vision fundamentals with the famous MNIST data <i>Getting Started</i> · Ongoing · tabular data, image data, multiclass classification, object identification	Knowledge 3,008 teams
---	---	--------------------------

The Bigger Picture

John Urbanic

Parallel Computing Scientist
Pittsburgh Supercomputing Center

Building Blocks

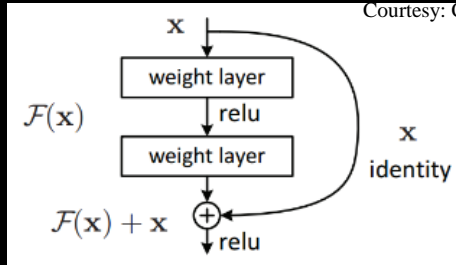
So far, we have used Fully Connected and Convolutional layers. These are ubiquitous, but there are many others:

- Fully Connected (FC)
- Convolutional (CNN)
- Residual (ResNet) [Feed forward]
- Recurrent (RNN), [Feedback, but has vanishing gradients so...]
- Long Short Term Memory (LSTM)
- Transformer (Attention based)
- Bidirectional RNN
- Restricted Boltzmann Machine
-
-

Several of these are particularly common...

Residual Neural Nets

We've mentioned that disappearing gradients can be an issue, and we know that deeper networks are more powerful. How do we reconcile these two phenomenae? One, very successful, method is to use some feedforward.



- Helps preserve reasonable gradients for very deep networks
- Very effective at imagery
- Used by AlphaGo Zero (40 residual CNN layers) in place of previous complex dual network
- 100s of layers common, Pushing 1000

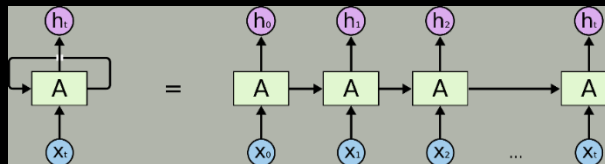
```
#Example: input 3-channel 256x256 image
x = Input(shape=(256, 256, 3))
y = Conv2D(3, (3, 3))(x)
z = keras.layers.add([x, y])
```

Haven't all of our Keras networks been built as strict layers in a *sequential* method? Indeed, but Keras supports a *functional* API that provides the ability to define network that branch in other ways. It is easy and here (<https://www.tensorflow.org/guide/keras/functional>) is an MNIST example with a 3 dense layers.

More to our current point, here (<https://www.kaggle.com/yadavsarthak/residual-networks-and-mnist>) is a neat experiment that uses **15(!)** residual layers to do MNIST. Not the most effective approach, but it works and illustrates the concept beautifully.

Recurrent Networks (RNNs)

If feedforward is useful, is there a place for feedback? Indeed, it is currently at the center of the many of the most effective techniques in deep learning.



Many problems occur in some context. Our MNIST characters are just pulled from a hat. However most character recognition has some context that can greatly aid the interpretation, as suggested by the following - not quite true - text.

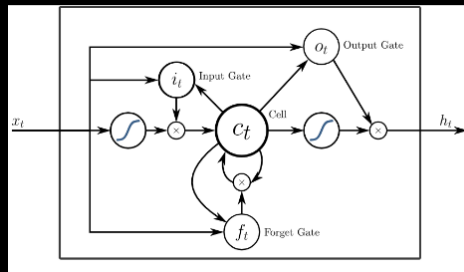
"Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mtttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteers be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe."

To pick a less confounding example. The following smudged character is pretty obvious by its context. If our network can "look back" to the previous words, it has a good chance at guessing the, otherwise unreadable, "a".

The dog chased the  at up the tree.

LSTMs

This RNN idea seems an awful lot like "memory", and suggests that we might actually incorporate a memory into networks. While the Long Short Term Memory (LSTM) idea was first formally proposed in 1997 by Hochreiter and Schmidhuber, it has taken on many variants since. This is often not explained and can be confusing if you aren't aware. I recommend "LSTM: A Search Space Odyssey" (Greff, et. al.) to help.



The basic design involves a memory cell, and some method of triggering a forget. `tf.keras.layers.LSTM` takes care of the details for us (but has a *lot* of options).

The Keras folks even provide us with an MNIST version (https://keras.io/examples/mnist_hierarchical_rnn/), although I think it is confusing as we are now killing a fly with a bazooka.

I recommend https://keras.io/examples/conv_lstm/, which uses network is used to predict the next frame of an artificially generated movie which contains moving squares. A much more natural fit.

Bi-directional LSTMs

Often, and especially in language processing, it is helpful to see both forward and backward. Take this example:

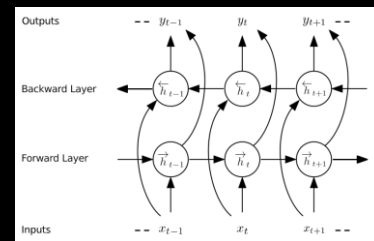
The dog chased the cat

Is the dog chasing a cat, or a car? If we read the rest of the sentence, it is obvious:

The dog chased the cat up the tree.

Adding even this very sophisticated type of network is easy in TF. Here is the network definition from the Keras *IMDB movie review sentiment analysis* example (https://www.tensorflow.org/tutorials/text/text_classification_rnn).

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(encoder.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1)
])
```



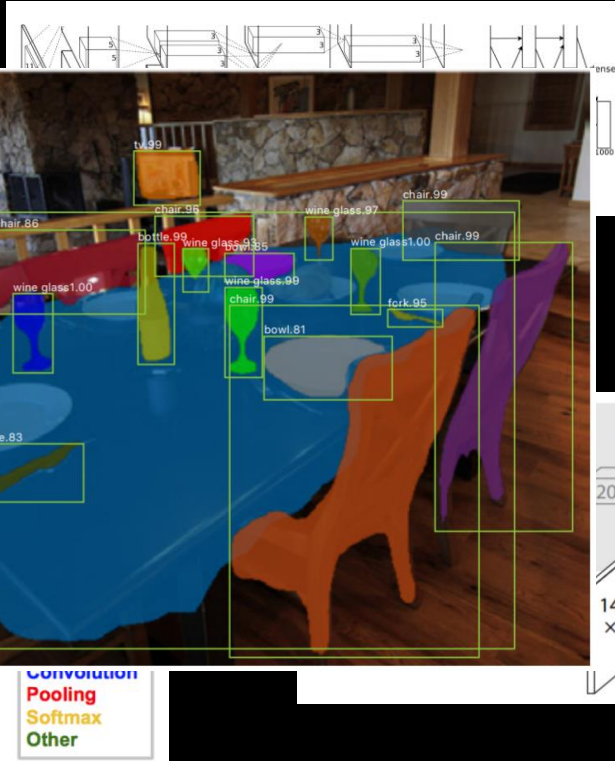
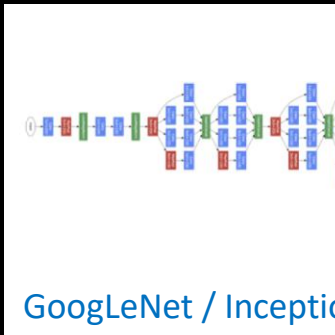
The first, embedding, layer introduces the concept of **word embeddings** - of central importance to any of you interested in natural language processing, and related to our running theme of **dimensionality reduction**. To oversimplify, here we are asking TF to reduce our vocabulary of vocab_size, so that every word's meaning is represented by a 64 dimensional vector.



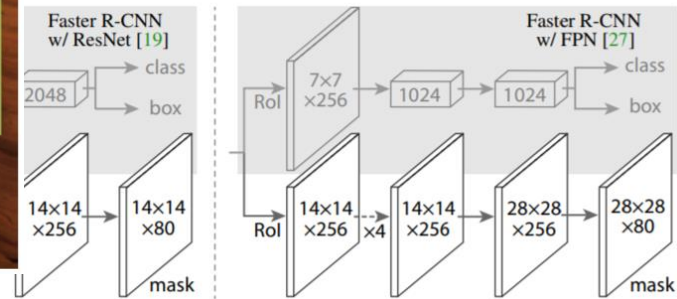
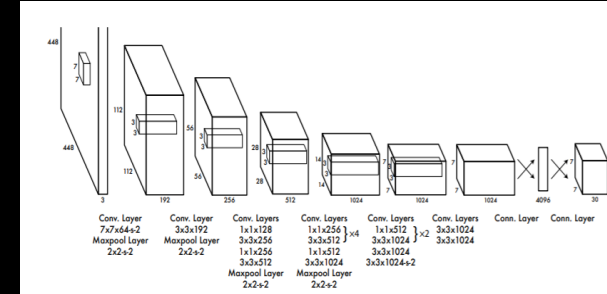
Architectures

With these layers, we can build countless different networks (and use TensorFlow to define them). Again, this is "3rd day" material, but we present them here and you should feel competent to research them yourself.

Generative Adversarial Network



YOLO (You Only Look Once)



Mask R-CNN

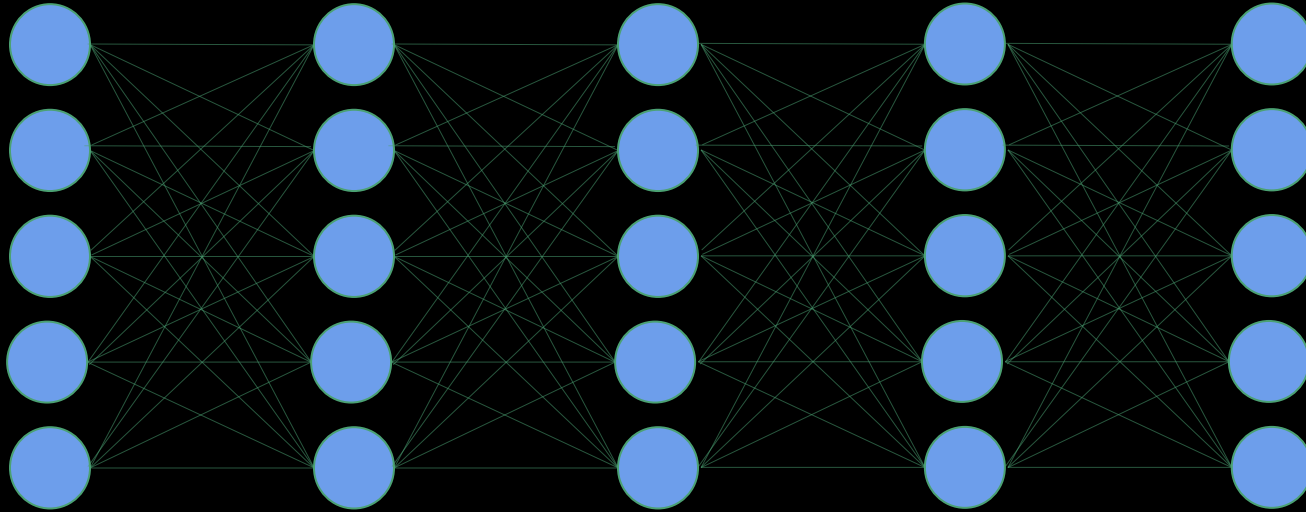
Autoencoder



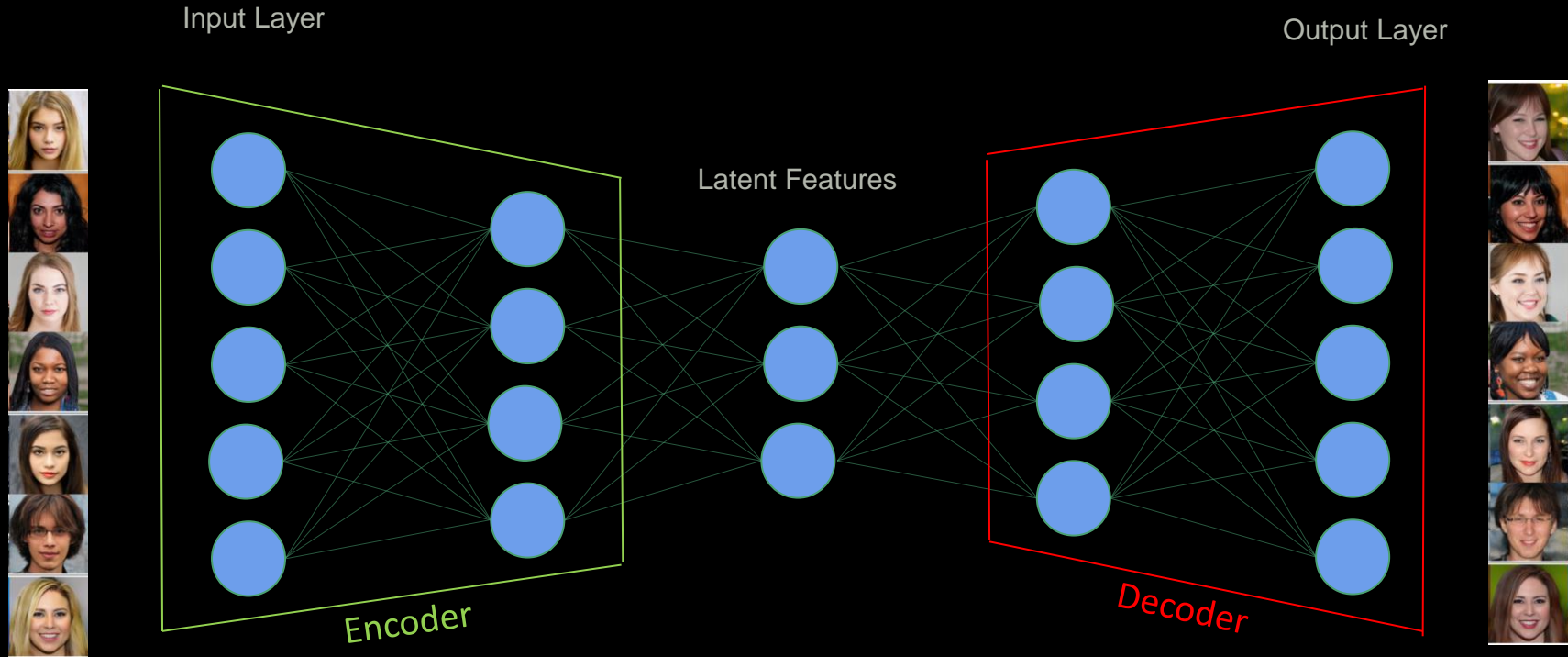
Input Layer

Hidden Layers

Output Layer



Autoencoder



Autoencoder



Input Layer

Output Layer

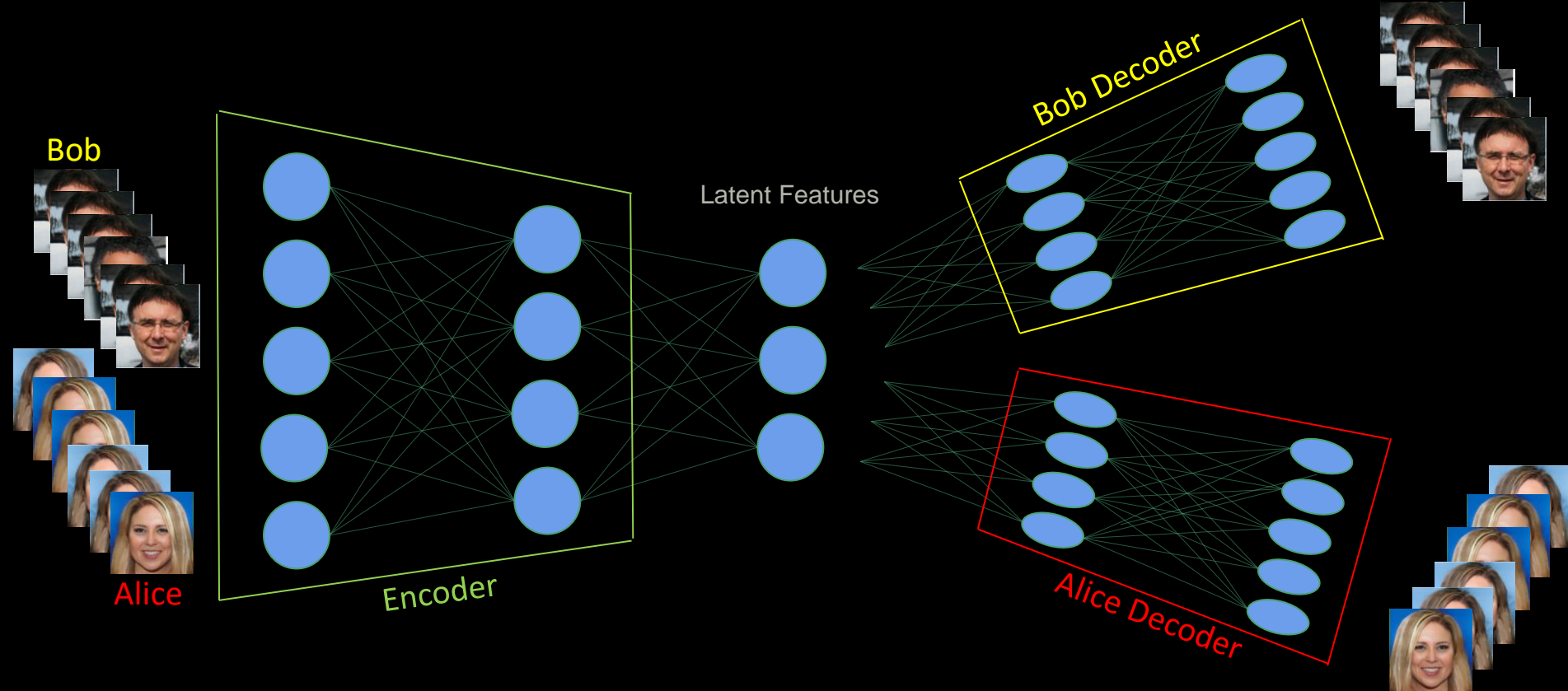
This autoencoder concept is very foundational.

It can be used for powerful *generational* networks by controlling the latent space as in *variational autoencoders*.

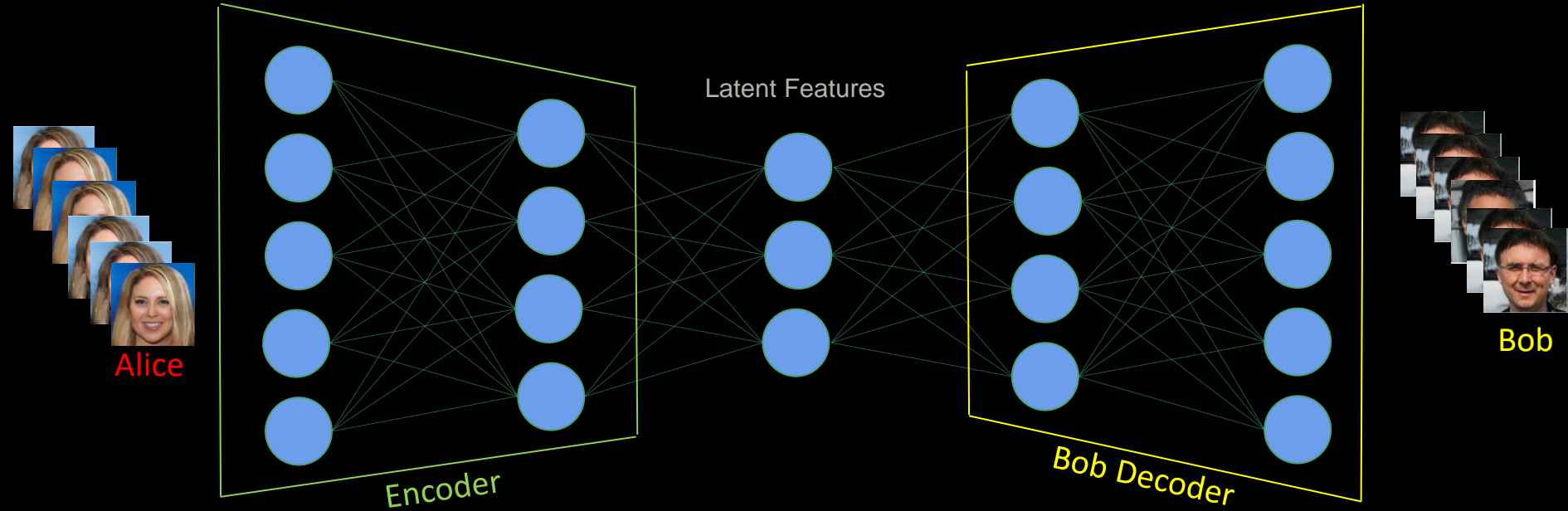
Or it can be a conceptual block in more complex designs like *transformers*.



Deepfake Training



Deepfake At Work



Zao Does DiCaprio

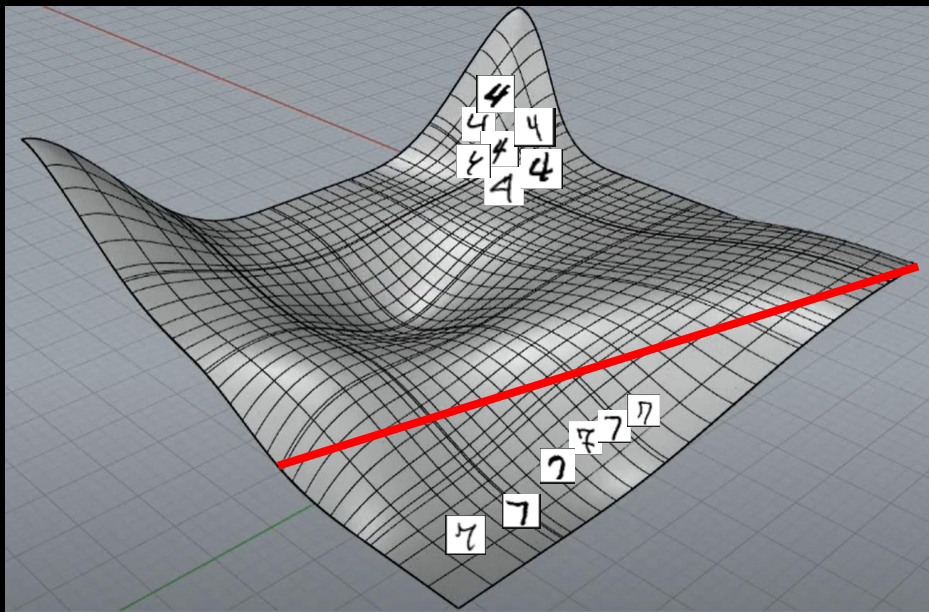
The Chinese app Zao did the below in 8 seconds from one photo.



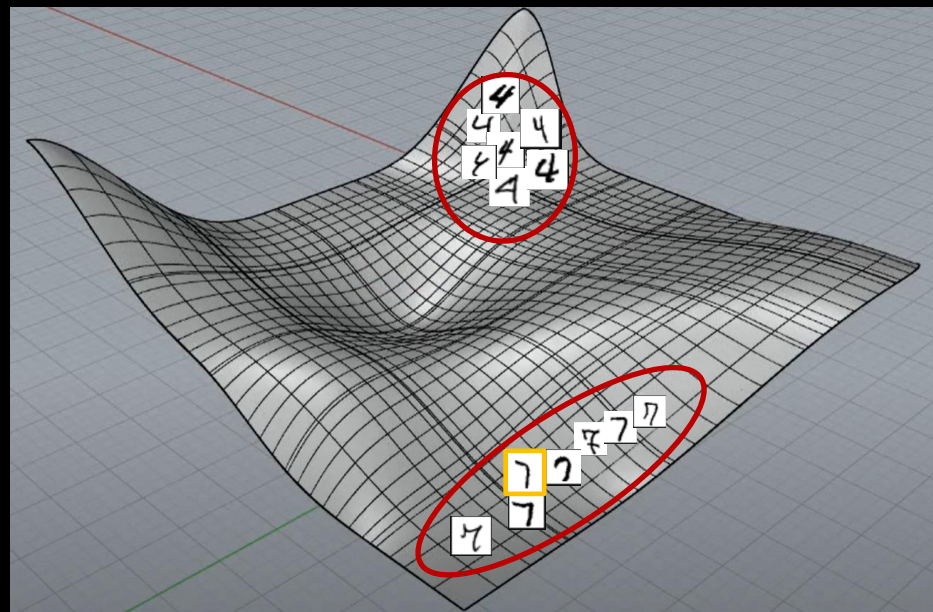
twitter.com/AllanXia/status/1168049059413643265

Discriminative vs. Generative

Discriminative models classify things, and need only know which side of the hyper-plane the instance lies on. Generative models need to understand the distribution to generate new instances.



Discriminative



Generative

Discriminative models need only capture the conditional probability of digit Y , given image X : $P(Y|X)$. Generative models must understand the joint probability $P(X,Y)$.

Other Tasks And Their Architectures

So far we have focused on images, and their classification. You know that deep learning has had success across a wide, and rapidly expanding, number of domains. Even our digit recognition task could be more sophisticated:

- Classification (What we did)
- Localization (Where is the digit?)
- Detection (Are there digits? How many?)
- Segmentation (Which pixels are the digits?)

These tasks would call for different network designs. This is where our Day 3 would begin, and we would use some other building blocks.

We don't have a Day 3, but we do have a good foundation to at least introduce the other important building blocks in current use.

Learning Approaches

Supervised Learning

How you learned colors.

What we have been doing just now.

Used for: image recognition, tumor identification, segmentation.

Requires labeled data.

Lots of it. Augmenting helps.

Reinforcement Learning

How you learned to walk.

Requires goals (maybe long term, i.e. arbitrary delays between action and reward).

Used for: Go (AlphaGo Zero), robot motion, video games.

Don't just read data, but *interact* with it!

Unsupervised Learning

(Maybe) how you learned to see.

What we did earlier with clustering and our recommender, and Deepfake.

Find patterns in data, compress data into model, find reducible representation of data.

Used for: Learning from unlabeled data.

All of these have been done with and without deep learning. DL has moved to the forefront of all of these.

AI Based Simulation?

A wise man once (not that long ago) told me "John, I don't need a neural net to rediscover conservation of energy."

Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach

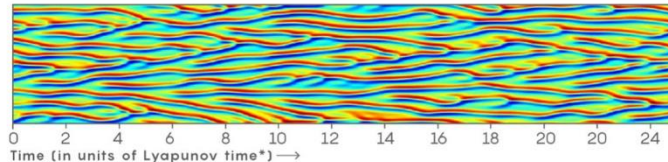
Jaideep Pathak, Brian Hunt, Michelle Girvan, Zhixin Lu, and Edward Ott
Phys. Rev. Lett. 120, 024102 – Published 12 January 2018

Training Computers to Tame Chaos

A machine-learning algorithm has been shown to accurately predict a chaotic system far further into the future than previously possible.

A Chaos Model

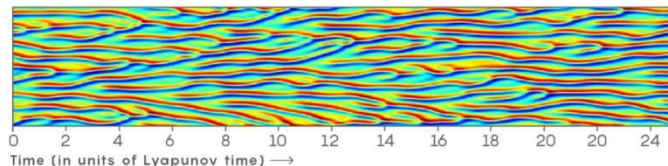
Researchers started with the evolving solution to the Kuramoto-Sivashinsky equation, which models propagating flames:



* Lyapunov time = Length of time before a small difference in the system's initial state begins to diverge exponentially. It typically sets the horizon of predictability, which varies from system to system.

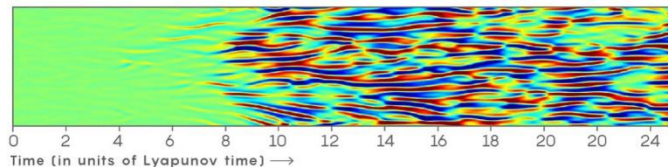
B Machine Learning

After training itself on data from the past evolution of the Kuramoto-Sivashinsky system, the "reservoir computing" algorithm predicts its future evolution:



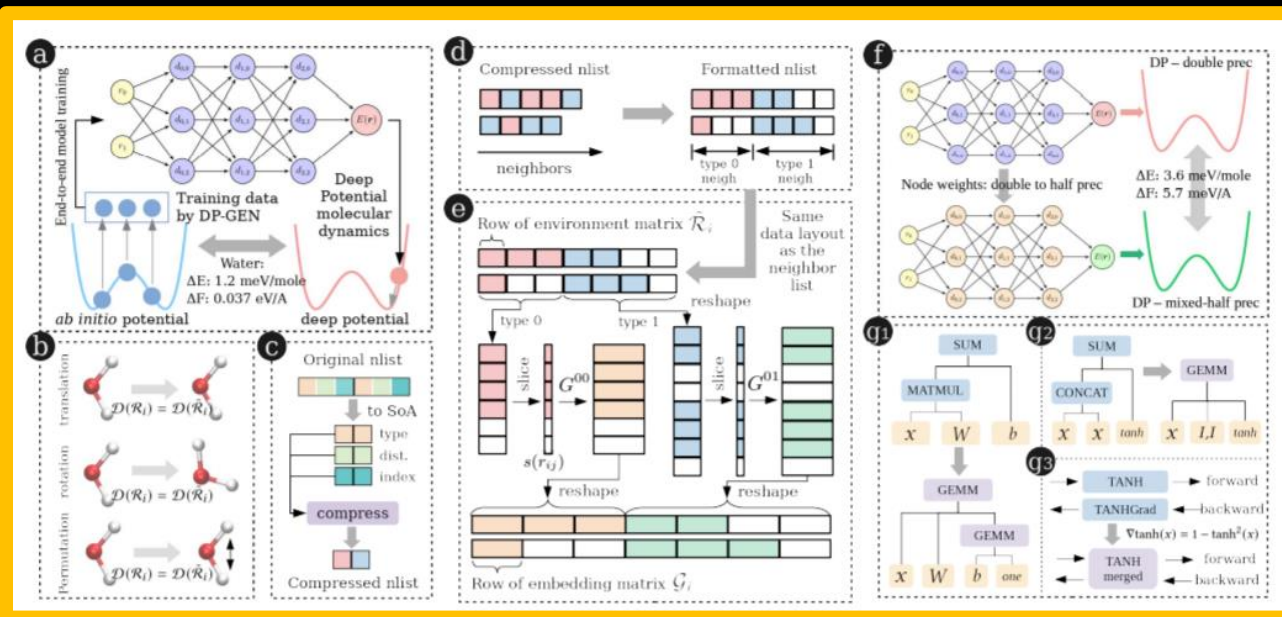
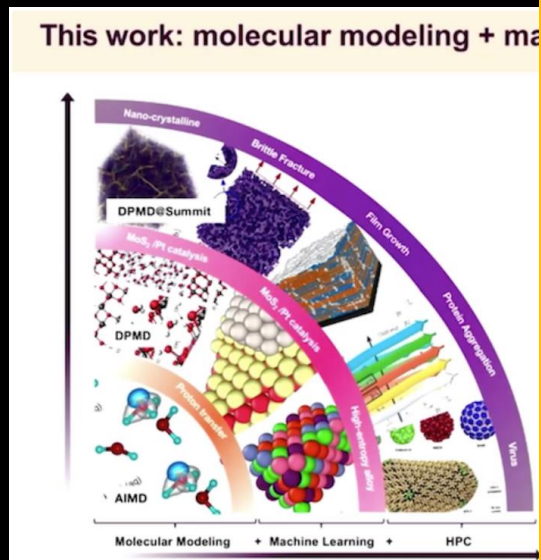
A - B Do They Match?

Subtracting B from A shows that the algorithm accurately predicts the model out to an impressive 8 Lyapunov times, before chaos ultimately prevails:



AI Based Simulation Is Here To Stay

“We report that a machine learning-based simulation protocol (Deep Potential Molecular Dynamics), while retaining *ab initio* accuracy, can simulate more than 1 nanosecond-long trajectory of over 100 million atoms per day, using a highly optimized code (GPU DeePMD-kit) on the Summit supercomputer. Our code can efficiently scale up to the entire Summit supercomputer, attaining 91 PFLOPS in double precision (45.5% of the peak) and 162/275 PFLOPS in mixed-single/half precision.



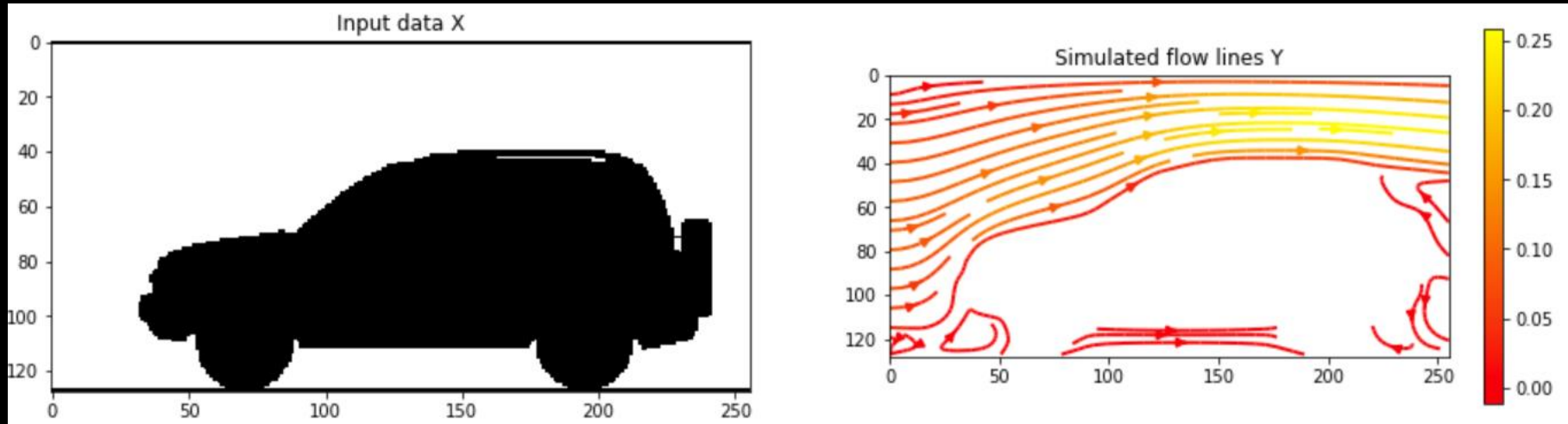
Pushing the Limit of Molecular Dynamics with Ab Initio Accuracy to 100 Million Atoms with Machine Learning

Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, Weinan E, Linfeng Zhang

2020 ACM Gordon Bell Prize Winner

Try It Yourself

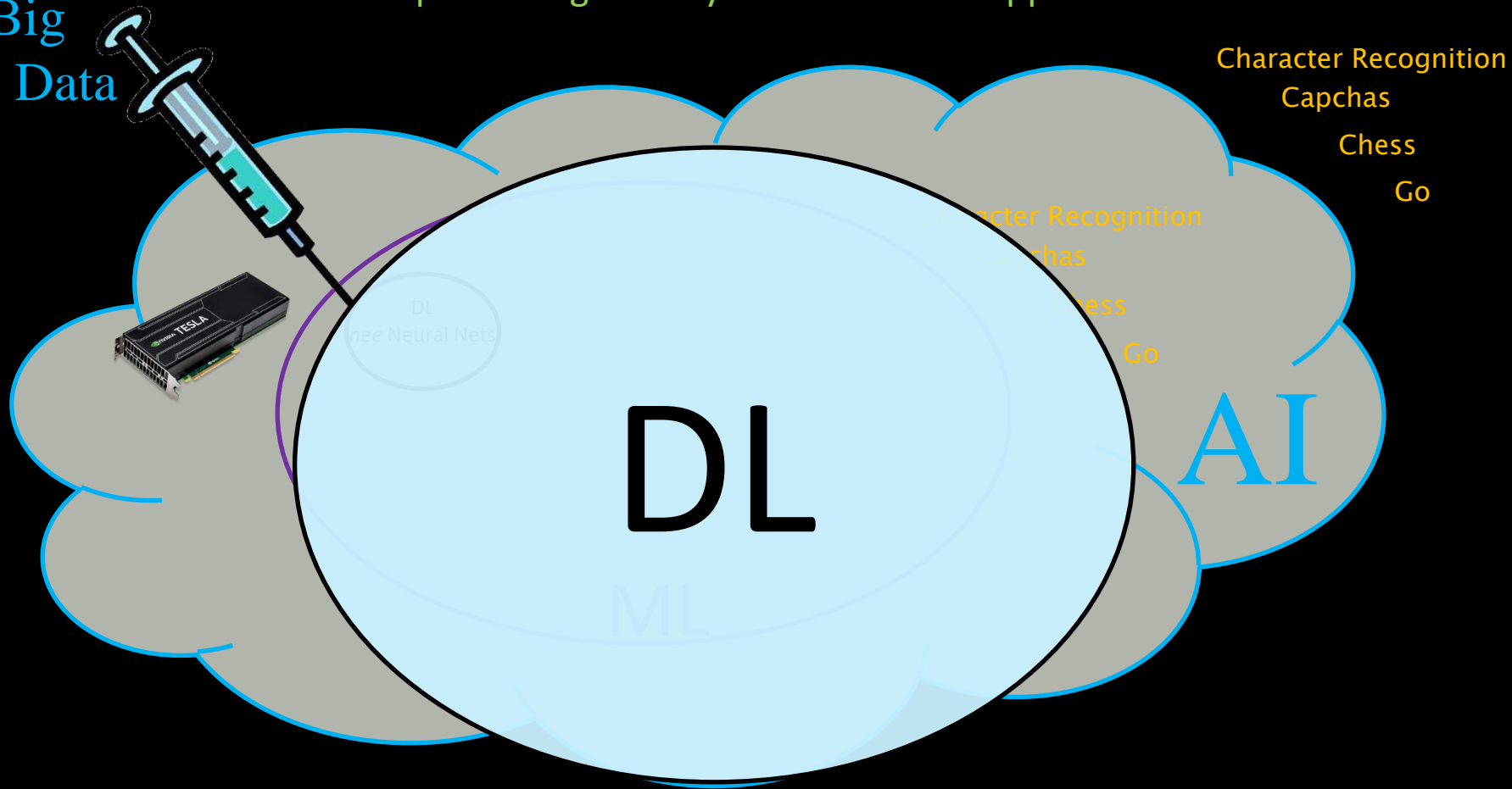
NVIDIA's GPU Bootcamp materials contain a great example of this type of work. The premise is to learn a mapping from boundary conditions to steady state fluid flow. The tutorial works through several different models, starting with a Fully Connected Network, then using a CNN and finally introducing a more advance Residual Network approach. You should be able to jump right in with what we have learned here.



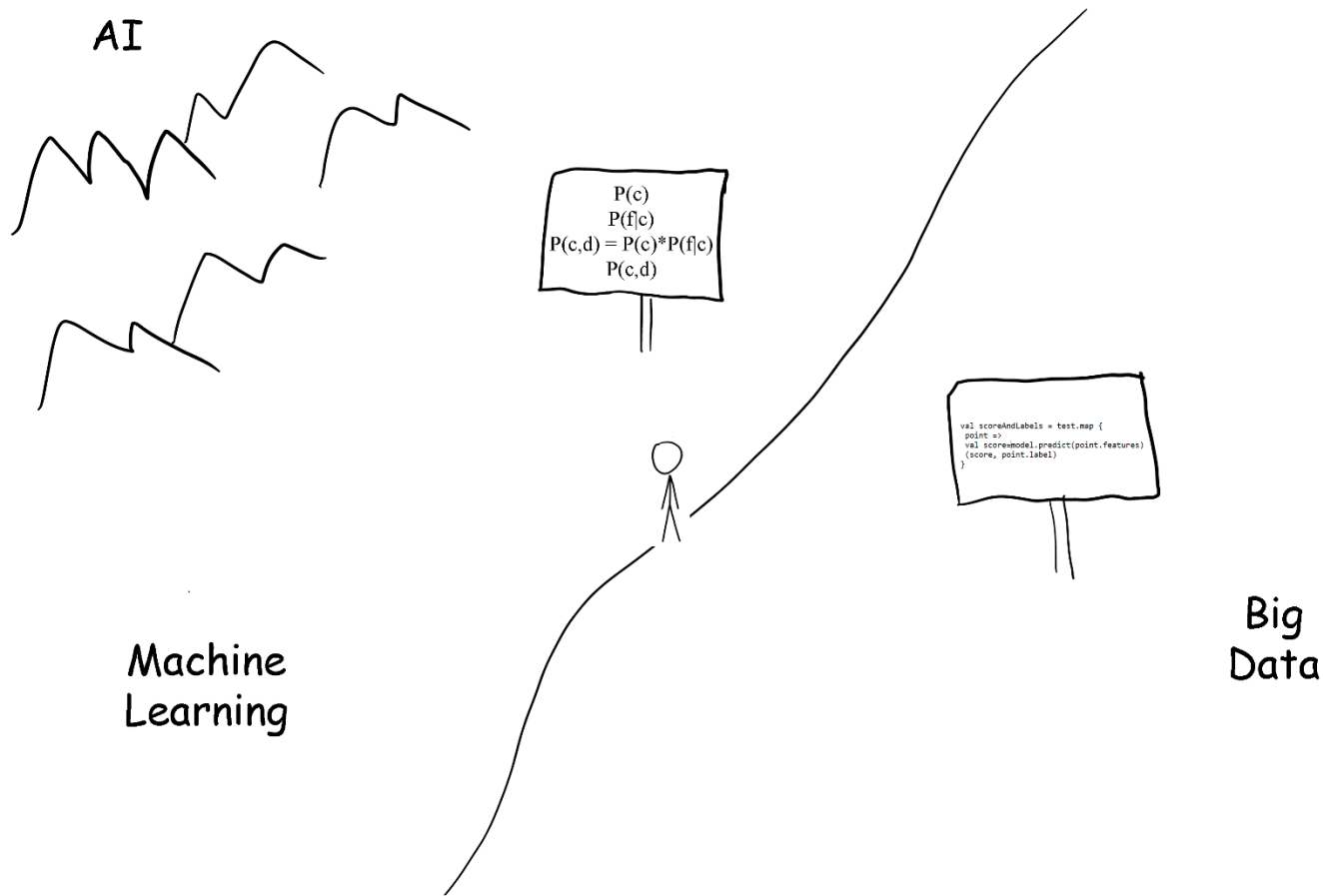
https://github.com/gpuhackathons-org/gpubootcamp/blob/78e9fee3432b60348489682a978fa63f29f7e839/hpc_ai/ai_science_cfd/English/python/jupyter_notebook/CFD/Start_Here.ipynb

Big
Data

Has Deep Learning left any room for other approaches?



Does
it
all
lead
to
Deep
Learning?



As the Data Scientist wanders across the ill-defined boundary between Data Science and Machine Learning, in search of the fabled land of Artificial Intelligence, they find that the language changes from programming to a creole of linear algebra and probability and statistics.

“Theoretician’s Nightmare” and Other Perspectives

The above is paraphrasing Yann LeCun, the godfather of Deep Learning.

If it feels like this is an oddly empirical branch of computer science, you are spot on.

Many of these techniques were developed through experimentation, and many of them are not amenable to classical analysis. A theoretician would suggest that non-convex loss functions are at the heart of the matter, and that situation isn’t getting better, as many of the latest techniques have made this much worse.

You may also have noticed that many of the techniques we have used today have very recent provenance. This is true throughout the field. Rarely is the undergraduate researcher so reliant upon groundbreaking papers of a few years ago.

The previously mentioned Christopher Olah has this rather useful summation: *"People sometimes complain: 'Neural networks are so hard to understand! Why can't we use understandable models, like SVMs?' Well, you understand SVMs, and you don't understand visual pattern recognition. If SVMs could solve visual pattern recognition, you would understand it. Therefore, SVMs are not capable of this, nor is any other model you can really understand."*

My own humble observation: Deep Learning looks a lot like late 19th century chemistry. There is a weak theoretical basis, but significant experimental breakthroughs of great utility. The lesson from that era was "expect a lot more perspiration than inspiration."

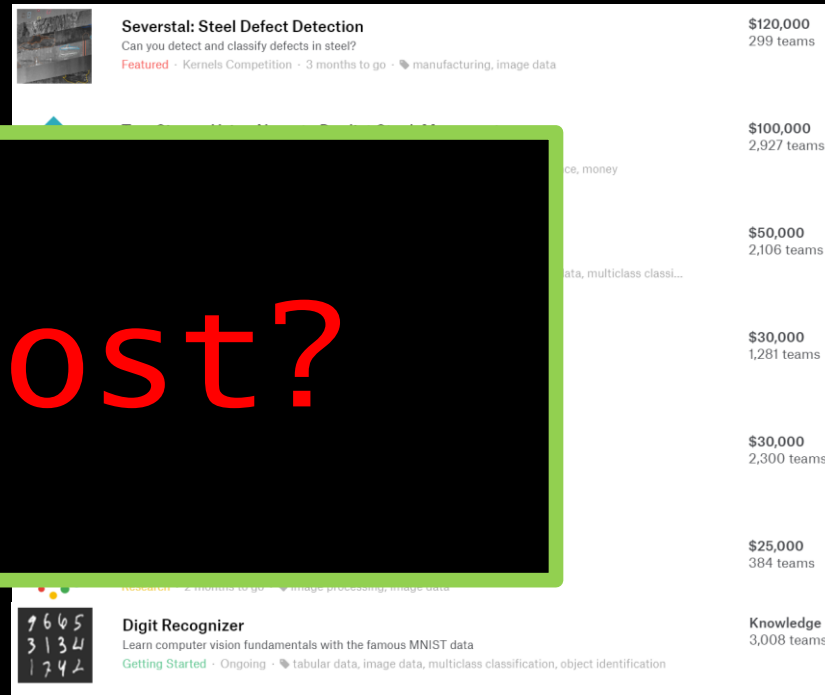
Lazy Scientist's Survey of the Field

Kaggle Challenge

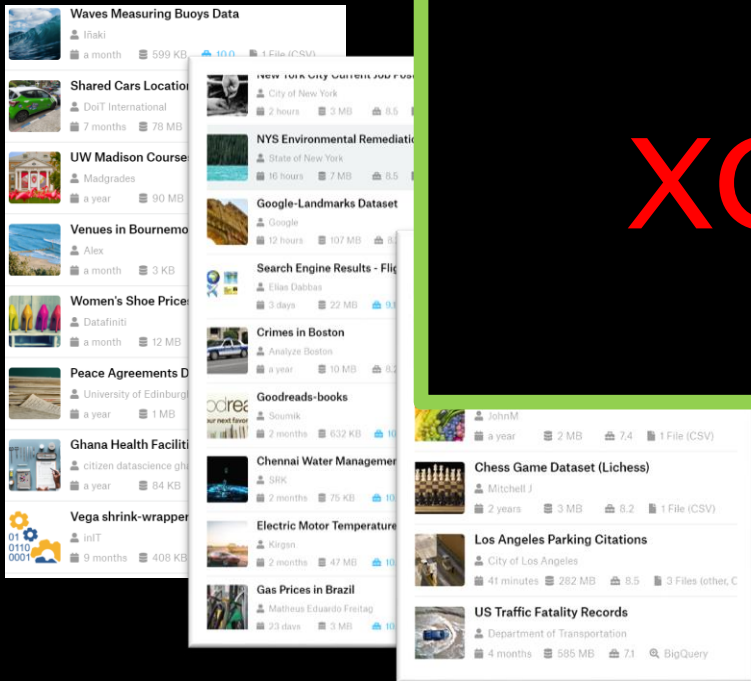
The benchmark driven nature of deep learning research, and its competitive consequences, have found a nexus at Kaggle.com. There you can find over 20,000 datasets:

and competitions:

XGBoost?



Competition Name	Prize Amount	Teams
Severstal: Steel Defect Detection	\$120,000	299 teams
...
...	\$100,000	2,927 teams
...	\$50,000	2,106 teams
...	\$30,000	1,281 teams
...	\$30,000	2,300 teams
...	\$25,000	384 teams
Digit Recognizer	Knowledge	3,008 teams

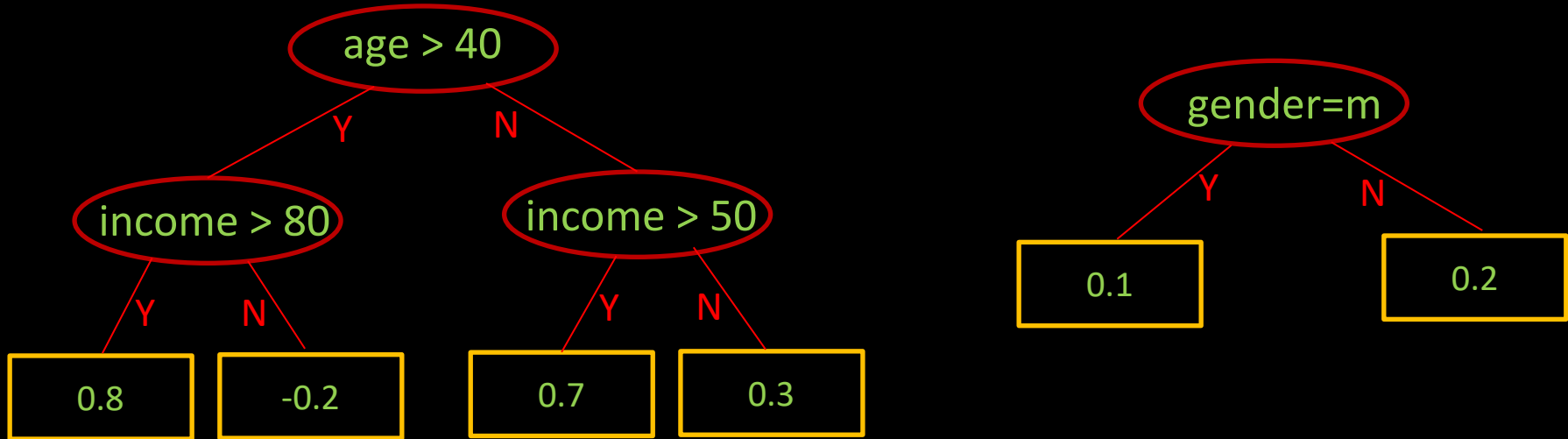


Dataset Name	Owner	Size	Files
Waves Measuring Buoys Data	Waki	599 KB	1 File (CSV)
Shared Cars Location	DoIT International	78 MB	
UW Madison Course	Madgrades	90 MB	
Venues in Bournemo	Alex	3 KB	
Women's Shoe Price	Datafiniti	12 MB	
Peace Agreements D	University of Edinburgh	1 MB	
Ghana Health Facilit	citizen datascience gha	84 KB	
Vega shrink-wrapper	0101100001	408 KB	
new york city census 2010	City of New York	3 MB	8.5
NYS Environmental Remediation	State of New York	7 MB	8.5
Google-Landmarks Dataset	Google	107 MB	8.5
Search Engine Results - Flickr	Elias Dabbas	22 MB	8.5
Crimes in Boston	Analyze Boston	10 MB	8.5
Goodreads-books	Soomik	632 KB	8.5
Chennai Water Management	SRK	75 KB	15
Electric Motor Temperature	Kingon	47 MB	15
Gas Prices in Brazil	Mathieu Eduardo Freitag	3 MB	15

Trees

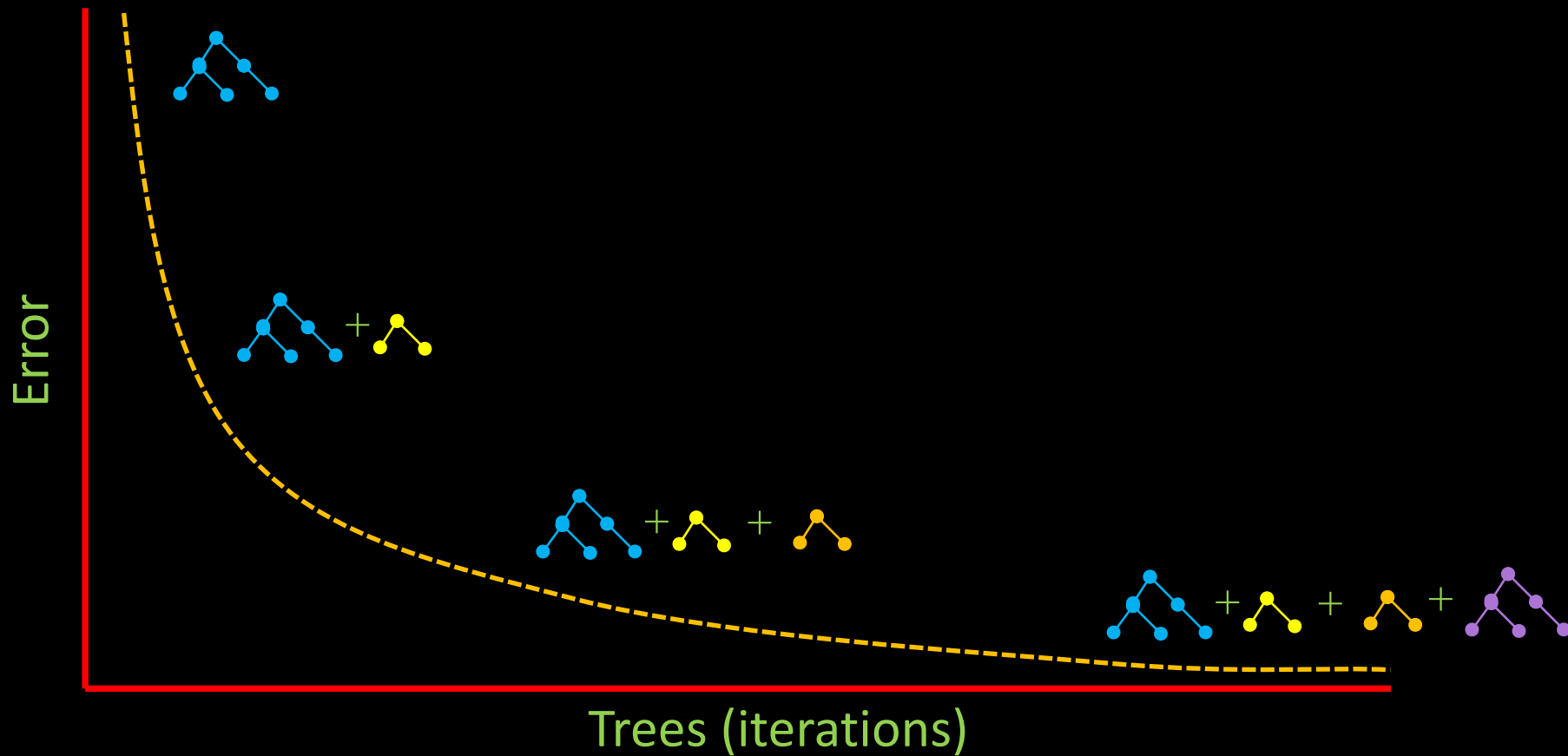
(How much of our earlier learning can we apply here?)

XGBoost is the latest, and most popular, evolution of the Decision Tree approach. Let's say we want to predict is some given person is likely to be a buyer of a certain car model:

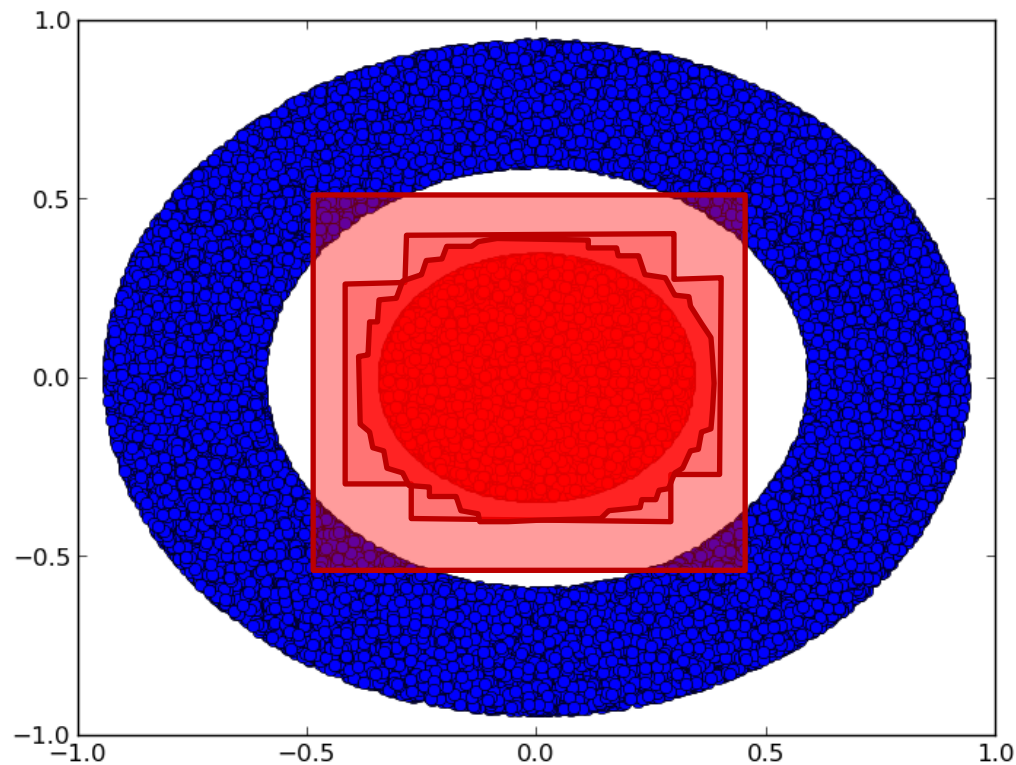


Trees are desirable in that they are non-linear, but still analytically tractable, and can do both regression and classification.

Gradient Boosted Trees

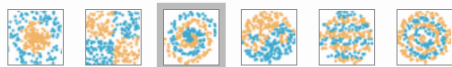


Remember This?

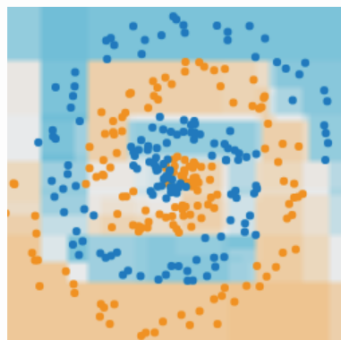


XGBoost

Dataset to classify:



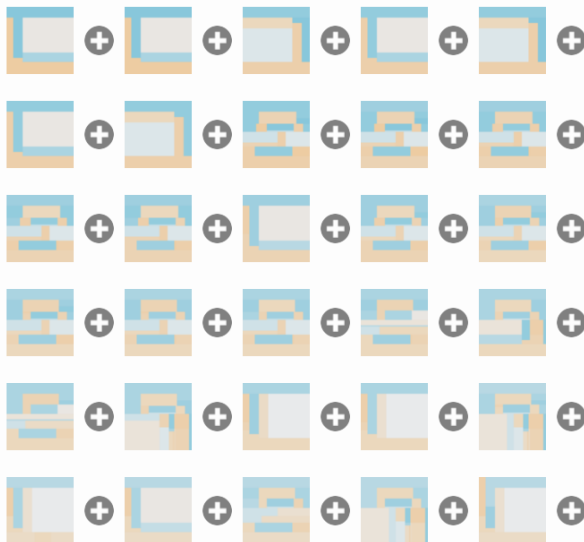
Prediction:



↑
predictions of GB (all 50 trees)

train loss: 0.381 test loss: 0.430

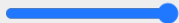
Decision functions of first 30 trees



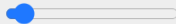
tree depth: 5



subsample: 100%



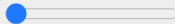
learning rate: 0.1



trees: 50



rotate dataset:



- ☐ rotate trees
- ☒ show gradients on hover
- ☐ use Newton-Raphson update

A very cool interactive application to explore these concepts, and try various hyperparameters, was done by Alex Rogozhnikov and can be found at:

http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html

If you want to understand XGBoost in detail, you can find the original paper at:

<https://arxiv.org/pdf/1603.02754.pdf>

An in-depth, but still beginner-friendly, video from StatsQuest can be found at:

<https://www.youtube.com/watch?v=GrJP9FLV3FE>

XGBoost in Particular

There are various implementations of gradient boosted trees. XGBoost combines several important innovations:

- Parallelizes well both across cores and nodes
- Clever cache optimization
- Works well with missing data

The end result is an efficient algorithm that works well enough with non-optimal hyperparameters the beginners can often make quick progress.

The scikit-learn version is probably the most popular, but there is a Spark version (https://xgboost.readthedocs.io/en/latest/jvm/xgboost4j_spark_tutorial.html), and if you want a deeper dive, NVIDIA has this pretty nice taxi fare regression model that uses GPUs with Spark and does a hyperparameter search. Note that I have not tried these myself:

<https://developer.nvidia.com/blog/accelerating-spark-3-0-and-xgboost-end-to-end-training-and-hyperparameter-tuning/>

TensorFlow has a boosted tree API along with a nice walkthrough example in the docs:

https://www.tensorflow.org/tutorials/estimator/boosted_trees

However, note that this is not the XGBoost version (yet).