

ML Design Pattern #3: Virtual Epochs

Base machine learning model training and evaluation on total number of examples, not on epochs or steps



Lak Lakshmanan

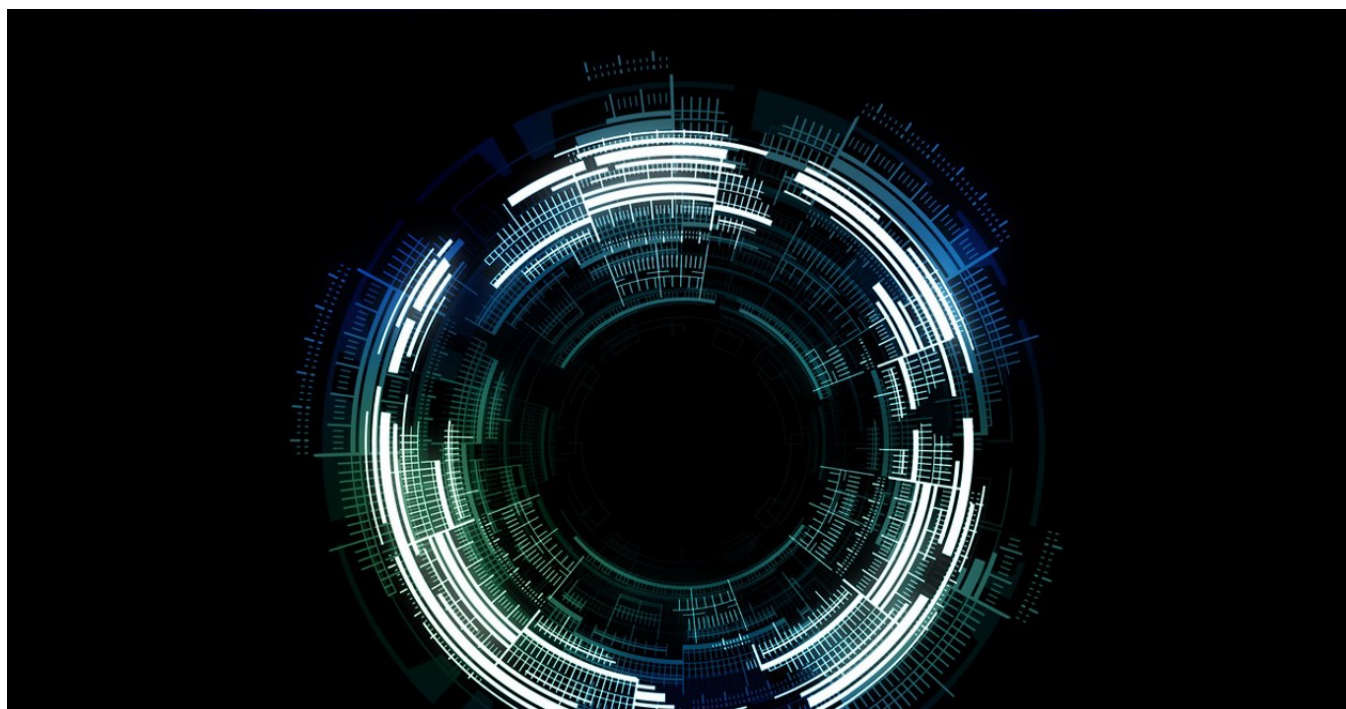
Sep 28, 2019 · 4 min read


An occasional series of design patterns for ML engineers. [Full list here.](#)

Machine learning tutorials often have code like [this](#):

```
model.fit(X_train, y_train,  
         batch_size=100,  
         epochs=15)
```

This code assumes that you have a dataset that fits in memory and can be iterated through `epochs` times without running the risk of machine failure. Both these assumptions are unreasonable — ML datasets range into terabytes and when training can last hours, the chances of machine failure are uncomfortably high.





When TensorFlow first came out, one of its selling points was that it didn't make such unreasonable assumptions. But the code necessary to build production-ready ML models was very hard to write. In TensorFlow 2.0, the API is much more intuitive. To make the above code more resilient, supply a `tf.dataset` (not just a numpy array) that provides iteration capability and lazy loading. Keras supports a callback that provides the ability to checkpoint (see [ML Design Pattern #2](#)). The code is now:

```
cp_callback = tf.keras.callbacks.ModelCheckpoint(...)
history = model.fit(trainds,
                    validation_data=evalds,
                    epochs=15,
                    batch_size=128,
                    callbacks=[cp_callback])
```

Epochs are problematic

However, using `epochs` is a bad idea. Epochs may be easy to understand, but the use of epochs leads to bad effects in real-world ML models. To see why, imagine that you have a training dataset with 1 million examples. It can be tempting to simply go through this dataset 15 times (for example) by setting the number of epochs to 15. There are several problems with this, though:

- The number of epochs is an integer, but the difference in training time between processing the dataset 14.3 times and 15 times can be hours. If the model has converged after having seen 14.3 million examples, you might want to exit and not waste the computational resources necessary to process 0.7 million more examples.
- You checkpoint once per epoch and waiting 1 million examples between checkpoints might be way too long. For resilience, you might want to checkpoint more often.
- Datasets grow over time. If you get 100,000 more examples and you train the model and get a higher error, is it because you need to do an early stop or is the new data corrupt in some way? You can't tell because the prior training was on 15 million examples and the new one is on 16.5 million examples.

Fixing the number of steps is not the answer

What if you fix the number of steps?

```
NUM_STEPS = 143000
BATCH_SIZE = 100
NUM_CHECKPOINTS = 15

cp_callback = tf.keras.callbacks.ModelCheckpoint(...)
history = model.fit(trains,
                    validation_data=evalds,
                    epochs=NUM_CHECKPOINTS,
                    steps_per_epoch=NUM_STEPS // NUM_CHECKPOINTS,
                    batch_size=BATCH_SIZE,
                    callbacks=[cp_callback])
```

Instead of training for 15 epochs of the original dataset, we are now training for 143,000 steps assuming that the batch size is 100. This gives us much more granularity, but we have to define an “epoch” as 1/15th of the total number of steps, so that we get the right number of checkpoints. This works as long as we make sure to repeat the `trains` infinitely:

```
trains = trains.repeat()
```

What happens when we get 100,000 more examples? Easy! We add it to our data warehouse but do not update the code. So, our code will still want to process 143,000 steps and it will get to process that much data. Except that 10% of the examples that it sees are newer. If the model converges, great. If it doesn't, we know that these new data points are the issue because we are not training longer than we were before! By keeping the number of steps constant, we have been able to separate out the effects of new data from training on more data.

Once we have trained for 143,000 steps, we restart the training and run it a bit longer (say 10,000 steps) and as long as the model continues to converge, we keep training it longer. Then, we update the number 143,000 in the code above (in reality, it will be a parameter to the code) to reflect the new number of steps.

This all works fine, until you want to do hyperparameter tuning. When you do hyperparameter tuning, you will want to want to change the batch size. Do you see what happens to the code above when you change the batch size? That's right — if you

change the batch size to 50, you will find yourself training for half the time! Obviously, this is no good.

Virtual epochs

The answer is to keep the total number of training examples shown to the model (not number of steps) constant:

```
NUM_TRAINING_EXAMPLES = 1000 * 1000
STOP_POINT = 14.3
TOTAL_TRAINING_EXAMPLES = int(STOP_POINT * NUM_TRAINING_EXAMPLES)
BATCH_SIZE = 100
NUM_CHECKPOINTS = 15

steps_per_epoch = (TOTAL_TRAINING_EXAMPLES //
                   (BATCH_SIZE*NUM_CHECKPOINTS))

cp_callback = tf.keras.callbacks.ModelCheckpoint(...)
history = model.fit(trainds,
                    validation_data=evalds,
                    epochs=NUM_CHECKPOINTS,
                    steps_per_epoch=steps_per_epoch,
                    batch_size=BATCH_SIZE,
                    callbacks=[cp_callback])
```

When you get more data, first train it with the old settings, then increase the number of examples to reflect the new data, then change the STOP_POINT to reflect the number of times you have to traverse the data to attain convergence.

This is now safe against hyperparameter tuning and retains all the advantages of keeping the number of steps constant.

Enjoy!

[Machine Learning](#)

[Design Patterns](#)

[ML Engineering](#)

[Keras](#)

[TensorFlow](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

