fast.ai

Making neural nets uncool again

Follow me of LinkedIn for more:
Steve Nouri
https://www.linkedin.com/in/stevenouri/

Our online courses (all are free and have no ads):

- Practical Deep Learning for Coders
- Part 2: Deep Learning from the Foundations
- Introduction to Machine Learning for Coders
- Computational Linear Algebra
- Code-First Introduction to Natural Language Processing

Our software: fastai v1 for PyTorch

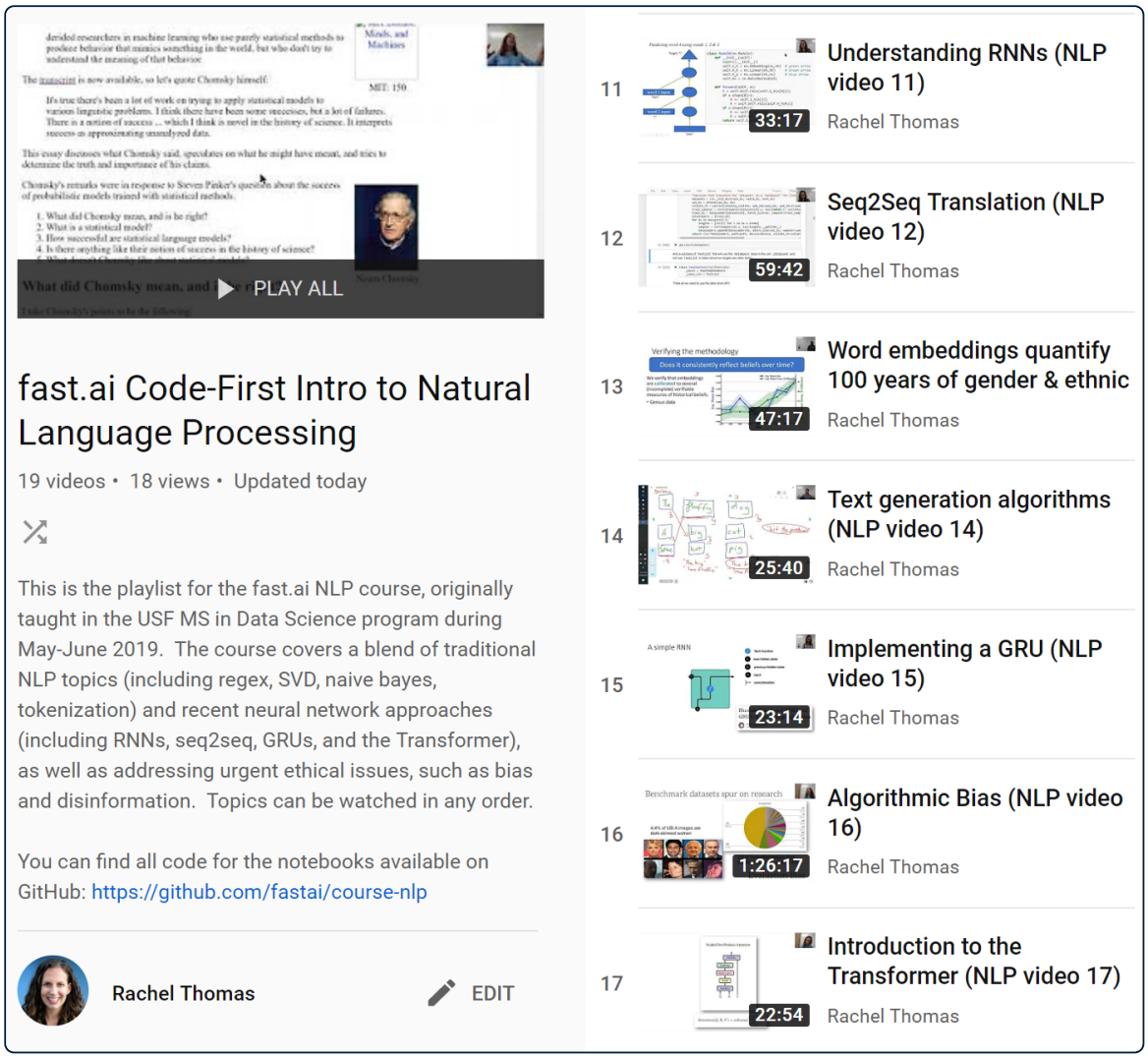
fast.ai in the news:

- The Economist: New schemes teach the masses to build AI
- MIT Tech Review: The startup diversifying AI workforce beyond just "techies"
- The New York Times: Finally, a Machine That Can Finish Your Sentence
- The Verge: An AI speed test shows clever coders can still beat tech giants like Google and Intel
- MIT Tech Review: A small team of student AI coders beats Google's machine-learning code
- Forbes: Artificial Intelligence Education Transforms The Developing World
- ZDNet: fast.ai's software could radically democratize AI

new fast.ai course: A Code-First Introduction to Natural Language Processing

08 Jul 2019 *Rachel Thomas*

Our newest course is a code-first introduction to NLP, following the fast.ai teaching philosophy of sharing practical code implementations and giving students a sense of the "whole game" before delving into lower-level details. Applications covered include topic modeling, classfication (identifying whether the sentiment of a review is postive or negative), language modeling, and translation. The course teaches a blend of traditional NLP topics (including regex, SVD, naive bayes, tokenization) and recent neural network approaches (including RNNs, seq2seq, attention, and the transformer architecture), as well as addressing urgent ethical issues, such as bias and disinformation. Topics can be watched in any order.



All videos for the course are on YouTube and all code is on GitHub

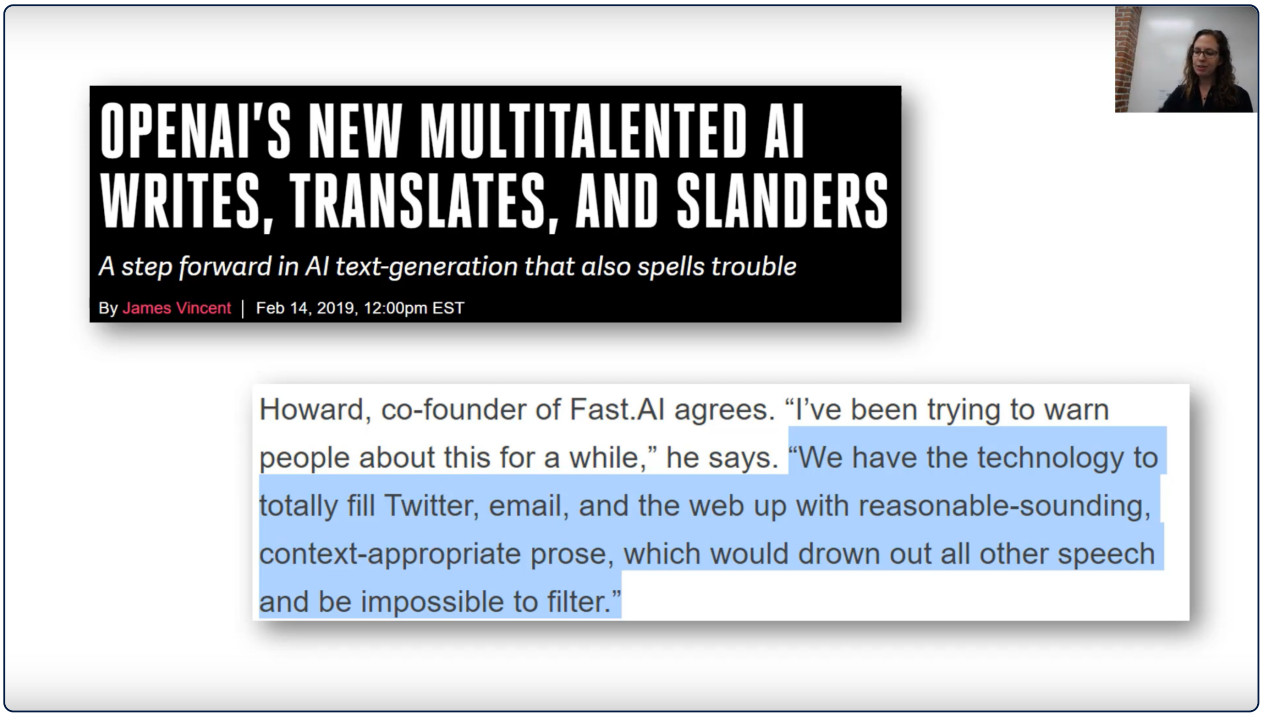
All the code is in Python in Jupyter Notebooks, using PyTorch and the fastai library. You can find all code for the notebooks available on GitHub and all the videos of the lectures are in this playlist.

This course was originally taught in the University of San Francisco MS in Data Science program during May-June 2019. The USF MSDS has been around for 7 years (over 330 students have graduated and gone on to jobs as data scientists during this time!) and is now housed at the Data Institute in downtown SF. In previous years, Jeremy taught the machine learning course and I've taught a computational linear algebra elective as part of the program.

Highlights

Some highlights of the course that I'm particularly excited about:

- Transfer learning for NLP
- Tips on working with languages other than English
- Attention and the Transformer
- Text generation algorithms (including the implementation of a new paper from the Allen Institute)
- Issues of bias and some steps towards addressing them
- A special guest lecture by Nikhil Garg on how word embeddings encode stereotypes (and how this has changed over the last 100 years)
- How NLP advances are heightening risks of disinformation



Risks raised by new language models such as GPT-2

Most of the topics can stand alone, so no need to go through the course in order if you are only interested in particular topics (although I hope everyone will watch the videos on bias and disinformation, as these are important topics for everyone interested in machine learning). Note that videos vary in length between 20-90 minutes.

Course Topics

Overview

There have been many major advances in NLP in the last year, and new state-of-the-art results are being achieved every month. NLP is still very much a field in flux, with best practices changing and new standards not yet settled on. This makes for an exciting time to learn NLP. This course covers a blend of more traditional techniques, newer neural net approaches, and urgent issues of bias and disinformation.

- What is NLP?

Traditional NLP Methods

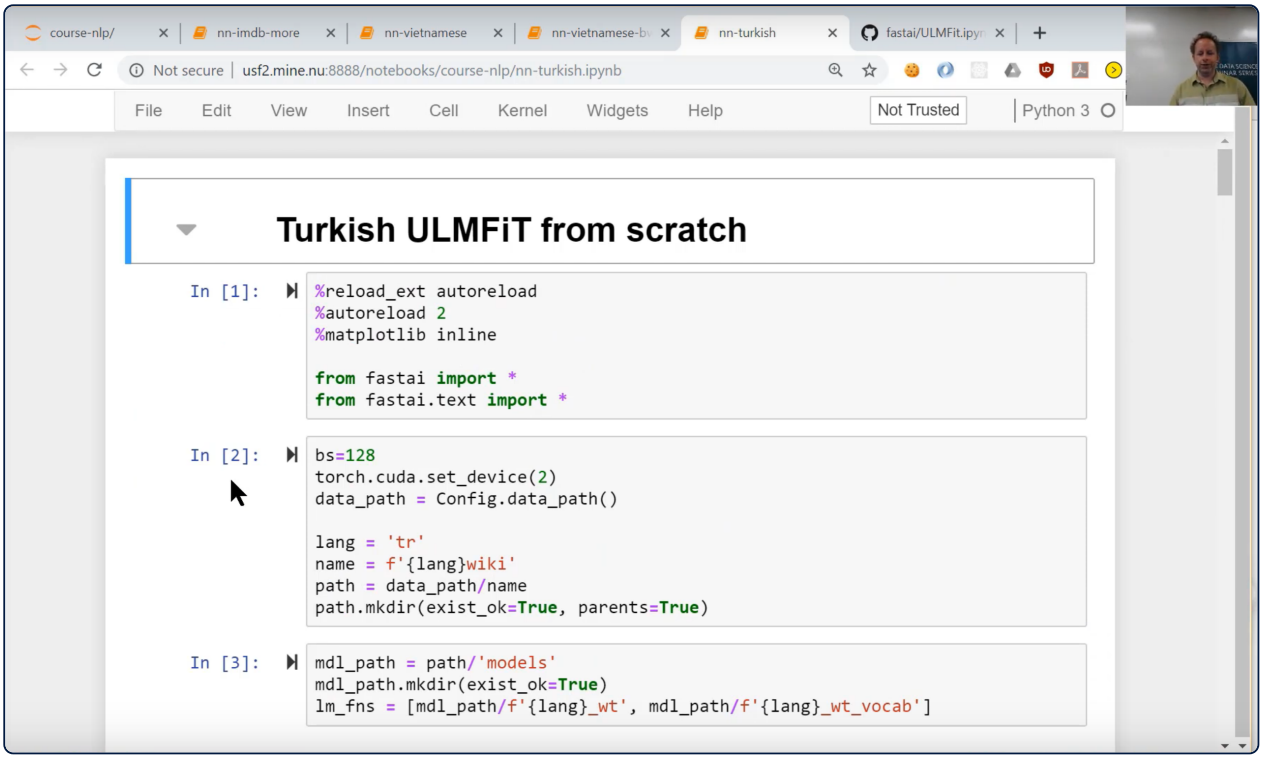
For the first third of the course, we cover topic modeling with SVD, sentiment classification via naive bayes and logisitic regression, and regex. Along the way, we learn crucial processing techniques such as tokenization and numericalizaiton.

- Topic Modeling with SVD & NMF
- Topic Modeling & SVD revisited
- Sentiment Classification with Naive Bayes
- Sentiment Classification with Naive Bayes & Logistic Regression, contd.
- Derivation of Naive Bayes, and Numerical Stability
- Revisiting Naive Bayes, and Regex

Deep Learning: Transfer learning for NLP

Jeremy shares jupyter notebooks stepping through ULMFit, his groundbreaking work with Sebastian Ruder last year to successfully apply transfer learning to NLP. The technique involves training a language model on a large corpus, fine-tuning it for a different and smaller corpus, and then adding a classifier to the end. This work has been built upon by more recent papers such as BERT, GPT-2, and XLNet. In new material (accompanying updates to the fastai library), Jeremy shares tips and tricks to work with languages other than English, and walks through examples implementing ULMFit for Vietnamese and Turkish.

- Intro to Language Modeling
- Transfer learning (Jeremy Howard)
- ULMFit for non-English Languages (Jeremy Howard)

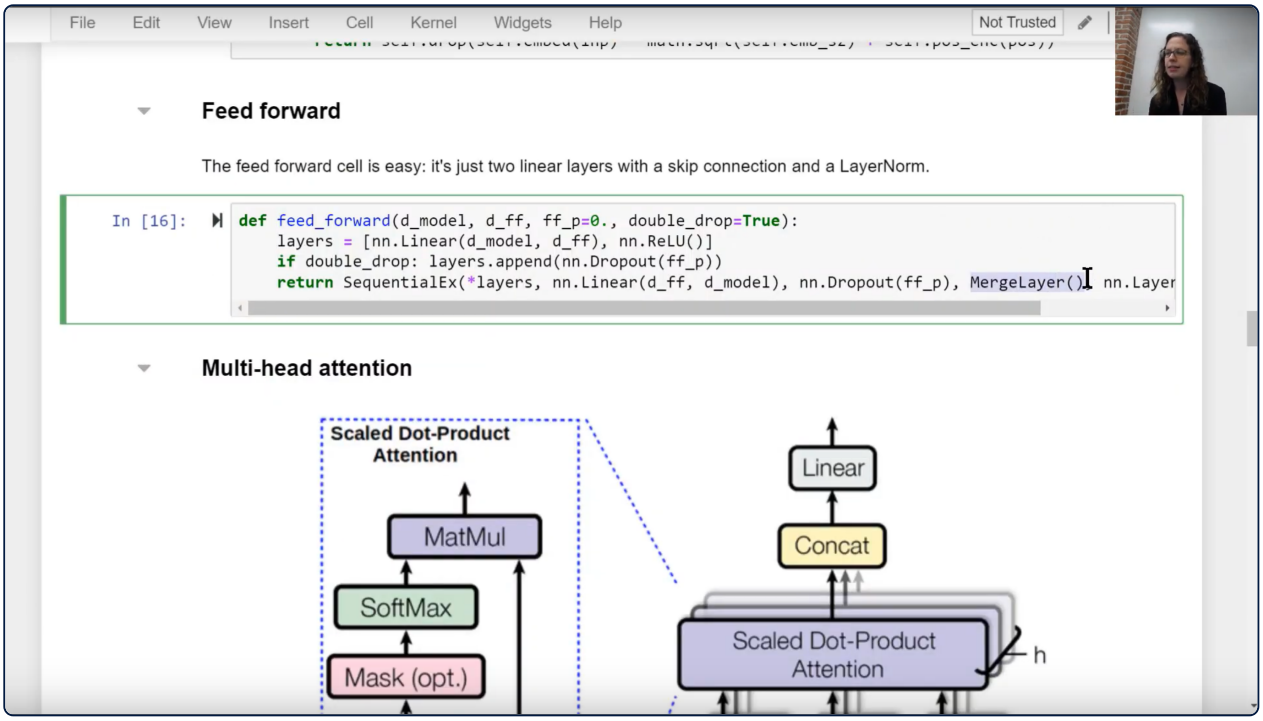


Jeremy shares ULMFit implementations in Vietnamese and Turkish

Deep Learning: Seq2Seq translation and the Transformer

We will dig into some underlying details of how simple RNNs work, and then consider a seq2seq model for translation. We build up our translation model, adding approaches such as teacher forcing, attention, and GRUs to improve performance. We are then ready to move on to the Transformer, exploring an implementation.

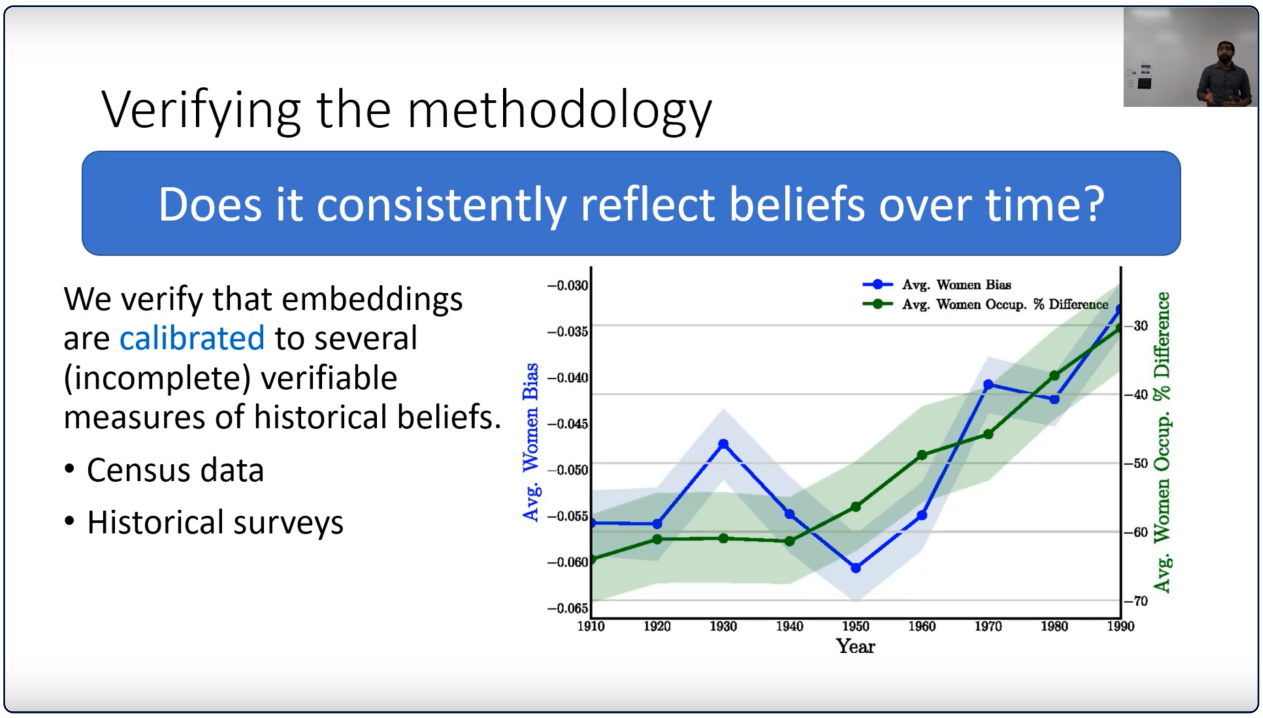
- Understanding RNNs
- Translation with Seq2Seq
- Text generation algorithms (Jeremy Howard)
- Implementing a GRU
- Introduction to the Transformer
- The Transformer for Language Translation



The Transformer for language translation

Ethical Issues in NLP

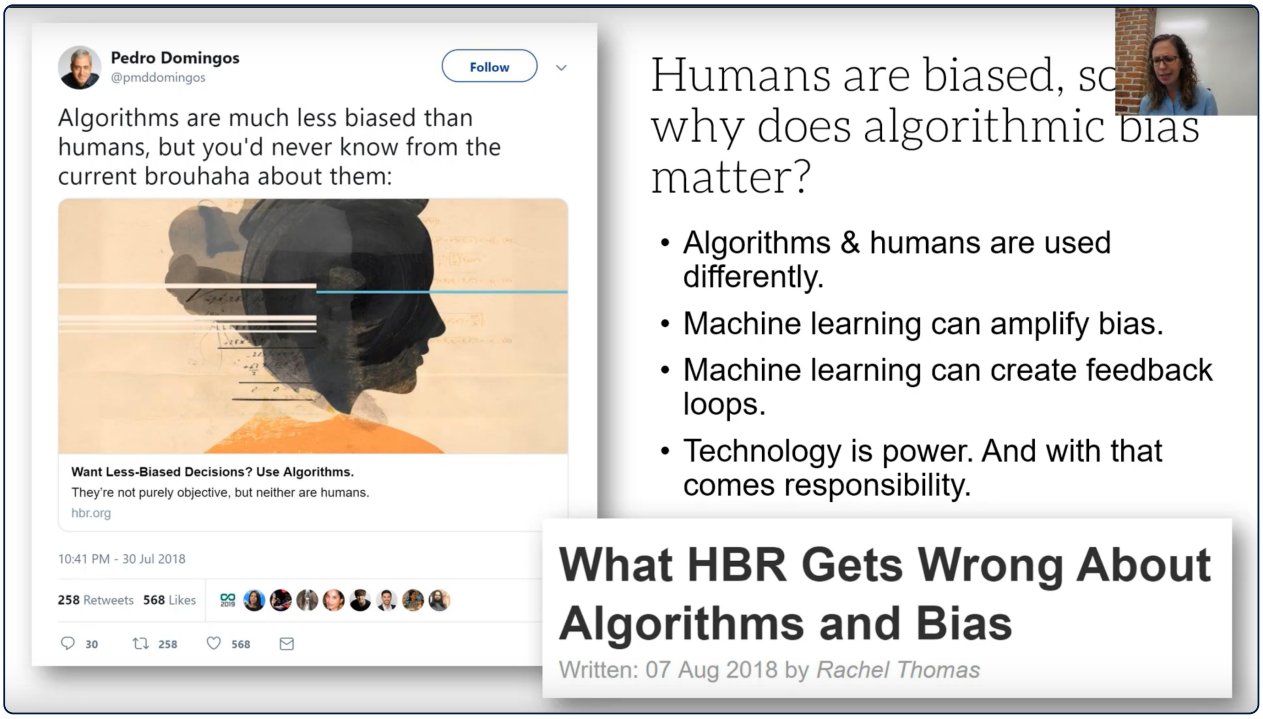
NLP raises important ethical issues, such as how stereotypes can be encoded in word embeddings and how the words of marginalized groups are often more likely to be classified as toxic. It was a special treat to have Stanford PhD student Nikhil Garg share his work which had been published in PNAS on this topic. We also learn about a framework for better understanding the causes of different types of bias, the importance of questioning what work we should avoid doing altogether, and steps towards addressing bias, such as Data Statements for NLP.



Nikhil Garg gave a guest lecture on his work showing how word embeddings quantify stereotypes over the last 100 years

Bias is not the only ethical issue in NLP. More sophisticated language models can create compelling fake prose that may drown out real humans or manipulate public opinion. We cover dynamics of disinformation, risks of compelling computer generated text, OpenAI's controversial decision of staged release for GPT-2, and some proposed steps towards solutions, such as systems for verification or digital signatures.

- Word embeddings quantify 100 years of gender & ethnic stereotypes (Nikhil Garg)
- Algorithmic Bias
- What you need to know about disinformation



On why algorithmic bias matters, different types, and steps towards addressing it

We hope you will check out the course! All the code for the jupyter notebooks used in the class can be found on GitHub and a playlist of all the videos is available on YouTube.

Prerequisites

(Updated to add) Familiarity with working with data in Python, as well as with machine learning concepts (such as training and test sets) is a necessary prerequisite. Some experience with PyTorch and neural networks is helpful.

As always, at fast.ai we recommend learning on an as-needed basis (too many students feel like they need to spend months or even years on background material before they can get to what really interests them, and too often, much of that background material ends up not even being necessary). **If you are interested in this course, but unsure whether you have the right background, go**

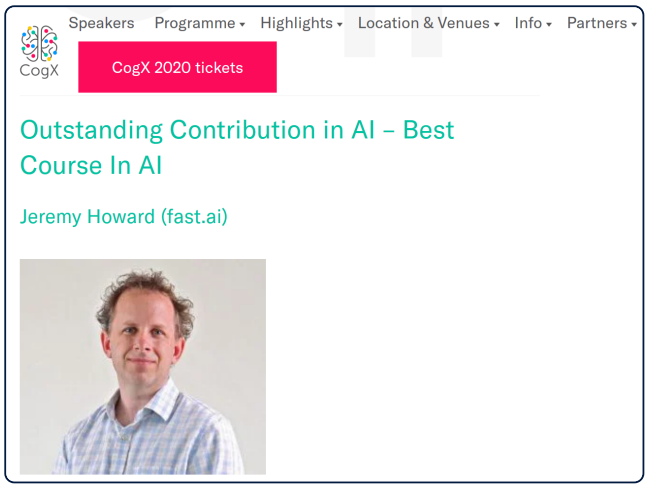
ahead and try the course! If you find necessary concepts that you are unfamiliar with, you can always pause and study up on them.

Also, please be sure to check out the fast.ai forums as a place to ask questions and share resources.

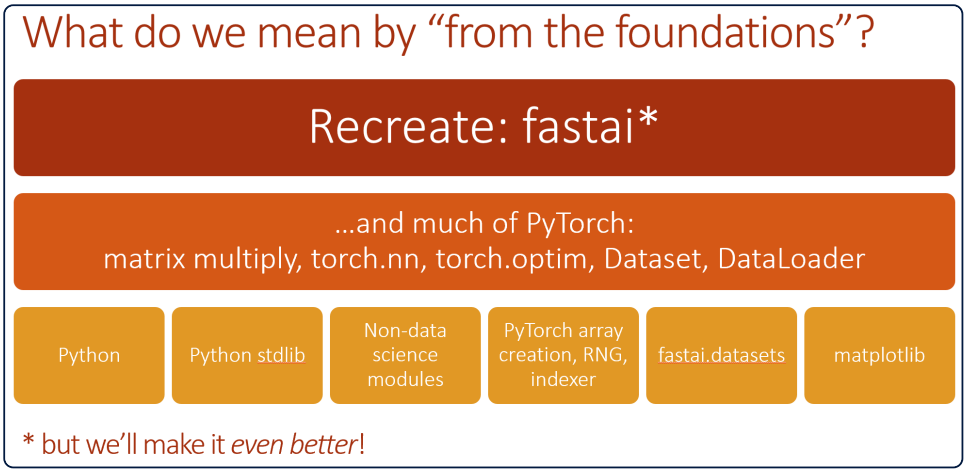
Deep Learning from the Foundations

28 Jun 2019 *Jeremy Howard*

Today we are releasing a new course (taught by me), Deep Learning from the Foundations, which shows how to build a state of the art deep learning model from scratch. It takes you all the way from the foundations of implementing matrix multiplication and back-propogation, through to high performance mixed-precision training, to the latest neural network architectures and learning techniques, and everything in between. It covers many of the most important academic papers that form the foundations of modern deep learning, using "*code-first*" teaching, where each method is implemented from scratch in python and explained in detail (in the process, we'll discuss many important software engineering techniques too). The whole course, covering around 15 hours of teaching and dozens of interactive notebooks, is entirely free (and ad-free), provided as a service to the community. The first five lessons use Python, PyTorch, and the fastai library; the last two lessons use Swift for TensorFlow, and are co-taught with Chris Lattner, the original creator of Swift, clang, and LLVM.



This course is the second part of fast.ai's 2019 deep learning series; part 1, Practical Deep Learning for Coders, was released in January, and is a required pre-requisite. It is the latest in our ongoing commitment to providing free, practical, cutting-edge education for deep learning practitioners and educators—a commitment that has been appreciated by hundreds of thousands of students, led to The Economist saying "*Demystifying the subject, to make it accessible to anyone who wants to learn how to build AI software, is the aim of Jeremy Howard… It is working*", and to CogX awarding fast.ai the *Outstanding Contribution in AI* award.



The purpose of *Deep Learning from the Foundations* is, in some ways, the opposite of part 1. This time, we're not learning practical things that we will use right away, but are learning foundations that we can build on. This is particularly important nowadays because this field is moving so fast. In this new course, we will learn to implement a lot of things that are inside the fastai and PyTorch libraries. In fact, we'll be reimplementing a significant subset of the fastai library! Along the way, we will practice implementing papers, which is an important skill to master when making state of the art models.

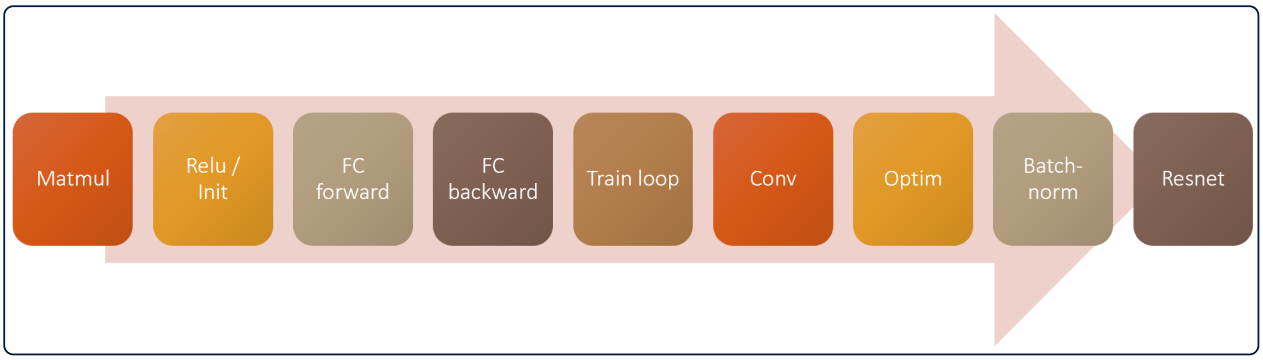
Chris Lattner at TensorFlow Dev Summit

A huge amount of work went into the last two lessons—not only did the team need to create new teaching materials covering both TensorFlow and Swift, but also create a new fastai Swift library from scratch, and add a lot of new functionality (and squash a few bugs!) in Swift for TensorFlow. It was a very close collaboration between Google Brain's Swift for TensorFlow group and fast.ai, and wouldn't have been possible without the passion, commitment, and expertise of the whole team, from both Google and fast.ai. This collaboration is ongoing, and today Google is releasing a new version of Swift for TensorFlow (0.4) to go with the new course. For more information about the Swift for TensorFlow release and lessons, have a look at this post on the TensorFlow blog.

In the remainder of this post I'll provide a quick summary of some of the topics you can expect to cover in this course—if this sounds interesting, then get started now! And if you have any questions along the way (or just want to chat with other students) there's a very active forum for the course, with thousands of posts already.

Lesson 8: Matrix multiplication; forward and backward passes

Our main goal is to build up to a complete system that can train Imagenet to a world-class result, both in terms of accuracy and speed. So we'll need to cover a lot of territory.



Our roadmap for training a CNN

Step 1 is matrix multiplication! We'll gradually refactor and accelerate our first, pure python, matrix multiplication, and in the process will learn about broadcasting and einstein summation. We'll then use this to create a basic neural net forward pass, including a first look at how neural networks are initialized (a topic we'll be going into in great depth in the coming lessons).

```
def matmul(a,b):
    (ar,ac),(br,bc) = a.shape,b.shape
    c = torch.zeros(ar, bc)
    for i in range(ar): c[i] = (a[i].unsqueeze(-1) * b).sum(dim=0)
    return c
```

```
%timeit -n 10 _=matmul(m1, m2)
```

```
325 µs ± 6 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
def matmul(a,b): return torch.einsum('ik,kj->ij', a, b)
```

```
%timeit -n 10 _=matmul(m1, m2)
```

```
69.6 µs ± 14.3 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Broadcasting and einsum let us accelate matmul dramatically

Then we will implement the backwards pass, including a brief refresher of the chain rule (which is really all the backwards pass is). We'll then refactor the backwards path to make it more flexible and concise, and finally we'll see how this translates to how PyTorch actually works.

```python
def relu_grad(inp, out): inp.g = (inp>0).float() * out.g
```

```python
def lin_grad(inp, out, w, b):
    inp.g = out.g @ w.t()
    w.g = (inp.unsqueeze(-1) * out.g.unsqueeze(1)).sum(0)
    b.g = out.g.sum(0)
```

```python
def forward_and_backward(inp, targ):
    l1 = inp @ w1 + b1
    l2 = relu(l1)
    out = l2 @ w2 + b2
    loss = mse(out, targ)
    mse_grad(out, targ)
    lin_grad(l2, out, w2, b2)
    relu_grad(l1, l2)
    lin_grad(inp, l1, w1, b1)
```

Back propagation from scratch

Papers discussed

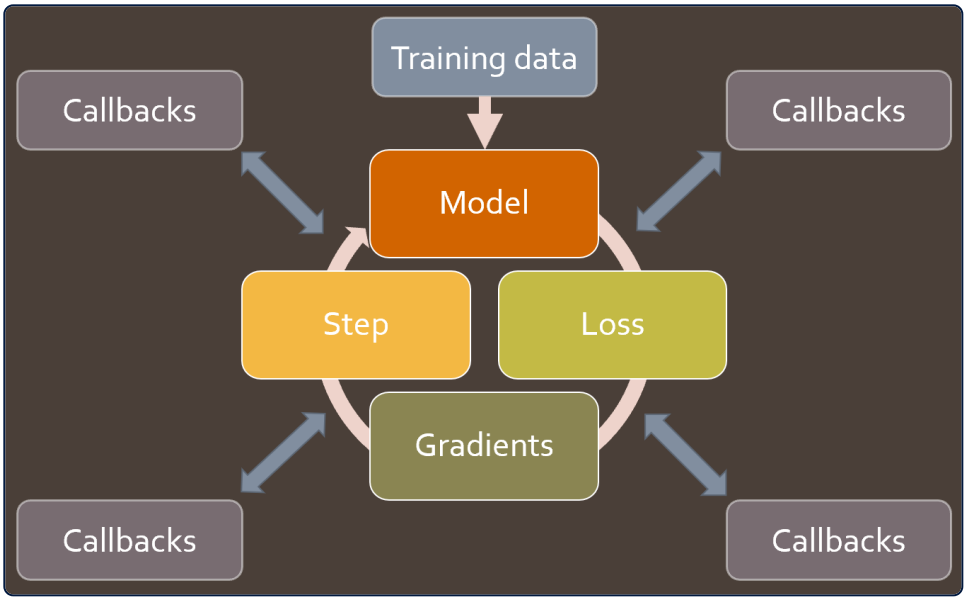
- Understanding the difficulty of training deep feedforward neural networks– paper that introduced Xavier initialization
- Fixup Initialization: Residual Learning Without Normalization – paper highlighting importance of normalisation - training 10,000 layer network without regularisation

Lesson 9: Loss functions, optimizers, and the training loop

In the last lesson we had an outstanding question about PyTorch's CNN default initialization. In order to answer it, I did a bit of research, and we start lesson 9 seeing how I went about that research, and what I learned. Students often ask "how do I do research", so this is a nice little case study.

Then we do a deep dive into the training loop, and show how to make it concise and flexible. First we look briefly at loss functions and optimizers, including implementing softmax and cross-entropy loss (and the *logsumexp* trick). Then we create a simple training loop, and refactor it step by step to make it more concise and more flexible. In the process we'll learn about `nn.Parameter` and `nn.Module`, and see how they work with `nn.optim` classes. We'll also see how `Dataset` and `DataLoader` really work.

Once we have those basic pieces in place, we'll look closely at some key building blocks of fastai: *Callback*, *DataBunch*, and *Learner*. We'll see how they help, and how they're implemented. Then we'll start writing lots of callbacks to implement lots of new functionality and best practices!



Callbacks in the training loop

Papers discussed

- Self-Normalizing Neural Networks (SELU)
- Exact solutions to the nonlinear dynamics of learning in deep linear neural networks (orthogonal initialization)
- All you need is a good init
- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification– 2015 paper that won ImageNet, and introduced ResNet and Kaiming Initialization.

Lesson 10: Looking inside the model

In lesson 10 we start with a deeper dive into the underlying idea of callbacks and event handlers. We look at many different ways to implement callbacks in Python, and discuss their pros and cons. Then we do a quick review of some other important foundations:

- `__dunder__` special symbols in Python
- How to navigate source code using your editor
- Variance, standard deviation, covariance, and correlation
- Softmax
- Exceptions as control flow

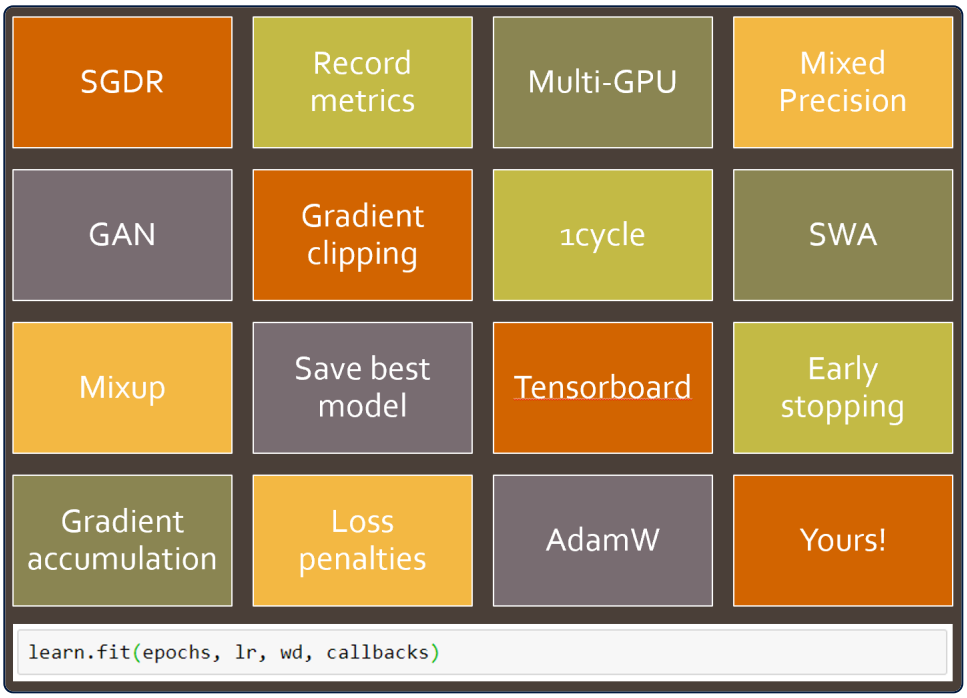
```
class SloppyAdder():
    def __init__(self,o): self.o=o
    def __add__(self,b): return SloppyAdder(self.o + b.o + 0.01)
    def __repr__(self): return str(self.o)
```

```
a = SloppyAdder(1)
b = SloppyAdder(2)
a+b
```

```
3.01
```

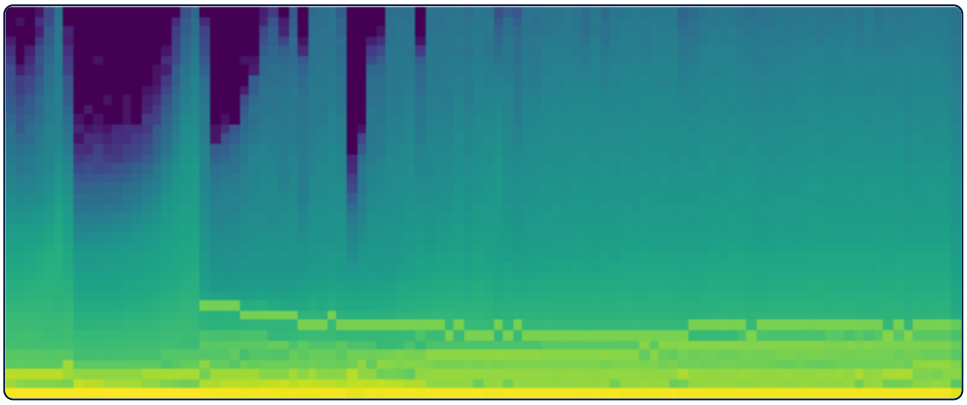
Python's special methods let us create objects that behave like builtin ones

Next up, we use the callback system we've created to set up CNN training on the GPU. This is where we start to see how flexible this sytem is—we'll be creating many callbacks during this course.



Some of the callbacks we'll create in this course

Then we move on to the main topic of this lesson: looking inside the model to see how it behaves during training. To do so, we first need to learn about *hooks* in PyTorch, which allow us to add callbacks to the forward and backward passes. We will use hooks to track the changing distribution of our activations in each layer during training. By plotting this distributions, we can try to identify problems with our training.



An example temporal activation histogram

In order to fix the problems we see, we try changing our activation function, and introducing batchnorm. We study the pros and cons of batchnorm, and note some areas where it performs poorly. Finally, we develop a new kind of normalization layer to overcome these problems, compare it to previously published approaches, and see some very encouraging results.

Papers discussed

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
- Layer Normalization
- Instance Normalization: The Missing Ingredient for Fast Stylization
- Group Normalization
- Revisiting Small Batch Training for Deep Neural Networks

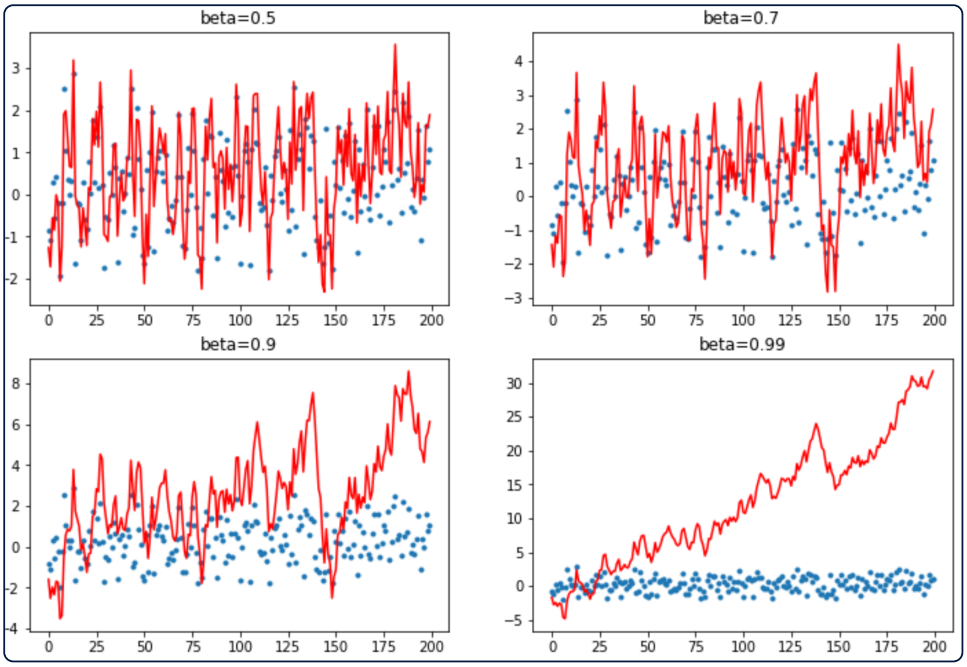
Lesson 11: Data Block API, and generic optimizer

We start lesson 11 with a brief look at a smart and simple initialization technique called Layer-wise Sequential Unit Variance (LSUV). We implement it from scratch, and then use the methods introduced in the previous lesson to investigate the impact of this technique on our model training. It looks pretty good!

Then we look at one of the jewels of fastai: the Data Block API. We already saw how to use this API in part 1 of the course; but now we learn how to create it from scratch, and in the process we also will learn a lot about how to better use it and customize it. We'll look closely at each step:

- Get files: we'll learn how `os.scandir` provides a highly optimized way to access the filesystem, and `os.walk` provides a powerful recursive tree walking abstraction on top of that
- Transformations: we create a simple but powerful `list` and function composition to transform data on-the-fly
- Split and label: we create flexible functions for each
- DataBunch: we'll see that `DataBunch` is a very simple container for our `DataLoader`s

Next up, we build a new `StatefulOptimizer` class, and show that nearly all optimizers used in modern deep learning training are just special cases of this one class. We use it to add weight decay, momentum, Adam, and LAMB optimizers, and take a look a detailed look at how momentum changes training.



The impact of varying momentum on a synthetic training example

Finally, we look at data augmentation, and benchmark various data augmentation techniques. We develop a new GPU-based data augmentation approach which we find speeds things up quite dramatically, and allows us to then add more sophisticated warp-based transformations.

```
%timeit -n 10 tfm_x = rotate_batch(x.cuda(), 128, 30)
```

```
1.92 ms ± 50.3 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Using GPU batch-level data augmentation provides big speedups

Papers discussed

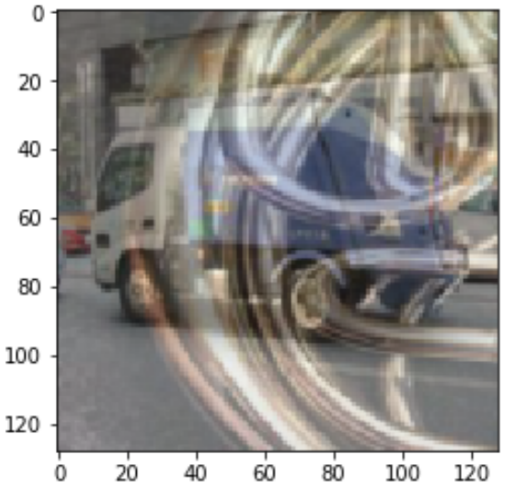
- L2 Regularization versus Batch and Weight Normalization
- Norm matters: efficient and accurate normalization schemes in deep networks
- Three Mechanisms of Weight Decay Regularization
- Nesterov's Accelerated Gradient and Momentum as approximations to Regularised Update Descent
- Adam: A Method for Stochastic Optimization
- Reducing BERT Pre-Training Time from 3 Days to 76 Minutes

Lesson 12: Advanced training techniques; ULMFiT from scratch

We implement some really important training techniques in lesson 12, all using callbacks:

- MixUp, a data augmentation technique that dramatically improves results, particularly when you have less data, or can train for a longer time
- Label smoothing, which works particularly well with MixUp, and significantly improves results when you have noisy labels
- Mixed precision training, which trains models around 3x faster in many situations.

```
mixed_up = ll.train.x[0]*0.5 + ll.train.x[4000]*0.5
plt.imshow(mixed_up.permute(1,2,0));
```

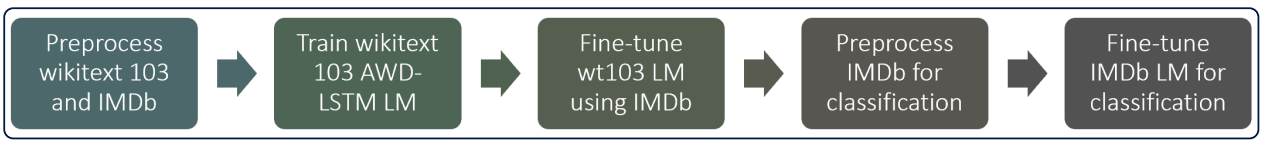


French horn or truck? The right answer is 50% french horn and 50% truck ;)

An example of MixUp augmentation

We also implement *xresnet*, which is a tweaked version of the classic resnet architecture that provides substantial improvements. And, even more important, the development of it provides great insights into what makes an architecture work well.

Finally, we show how to implement ULMFiT from scratch, including building an LSTM RNN, and looking at the various steps necessary to process natural language data to allow it to be passed to a neural network.



ULMFiT

Papers discussed

- mixup: Beyond Empirical Risk Minimization
- Rethinking the Inception Architecture for Computer Vision (label smoothing is in part 7)

- [Bag of Tricks for Image Classification with Convolutional Neural Networks]
- [Universal Language Model Fine-tuning for Text Classification]

Lesson 13: Basics of Swift for Deep Learning

By the end of lesson 12, we've completed building much of the fastai library for Python from scratch. Next we repeat the process for Swift! The final two lessons are co-taught by Jeremy along with Chris Lattner, the original developer of Swift, and the lead of the Swift for TensorFlow project at Google Brain.

Swift

```
struct MyModel: Layer {
   var conv = Conv2D<Float>(filterShape: (5, 5, 3, 6))
   var pool = MaxPool2D<Float>(2)
   var flatten = Flatten<Float>()
   var dense = Dense<Float>(16*5*5, 10)
   @differentiable
   func call(_ input: Tensor<Float>) -> Tensor<Float> {
       return dense(flatten(pool(conv(input)))) }}
```

Python

```
class MyModel(nn.Model):
   def __init__(self):
      super().__init__()
      self.conv = nn.Conv2d(3,6,kernel_size=5)
      self.pool = nn.MaxPool2d(2)
      self.flatten = Flatten()
      self.dense = nn.Linear(16*5*5, 10)
   def forward(self, x):
      return self.dense(self.flatten(self.pool(self.conv(input))))
```

Swift code and Python code don't look all that different

In this lesson, Chris explains what Swift is, and what it's designed to do. He shares insights on its development history, and why he thinks it's a great fit for deep learning and numeric programming more generally. He also provides some background on how Swift and TensorFlow fit together, both now and in the future. Next up, Chris shows a bit about using types to ensure your code has less errors, whilst letting Swift figure out most of your types for you. And he explains some of the key pieces of syntax we'll need to get started.

Chris also explains what a compiler is, and how LLVM makes compiler development easier. Then he shows how we can actually access and change LLVM builtin types directly from Swift! Thanks to the compilation and language design, basic code runs very fast indeed - about 8000 times faster than Python in the simple example Chris showed in class.

```
  public static var infinity: ${Self} {
%if bits == 32:
    return ${Self}(bitPattern: 0b0_11111111_00000000000000000000000)
%elif bits == 64:
    return ${Self}(
      bitPattern: 0b0_11111111111_0000000000000000000000000000000000000000000000000000)
%elif bits == 80:
    let rep = _Representation(
      explicitSignificand: ${Self}._explicitBitMask,
      signAndExponent: 0b0_111111111111111)
    return unsafeBitCast(rep, to: ${Self}.self)
%else:
    return ${Self}(sign: .plus,
      exponentBitPattern: _infinityExponent,
      significandBitPattern: 0)
%end
  }
```

Learning about the implementation of `float` in Swift

Finally, we look at different ways of calculating matrix products in Swift, including using Swift for TensorFlow's `Tensor` class.

Swift resources

- The swift book
- A swift tour (download in playground on an iPad or a Mac if you can).
- The harebrain forum category. This is where to ask your S4TF questions.
- Why fastai is embracing S4TF?

Lesson 14: C interop; Protocols; Putting it all together

Today's lesson starts with a discussion of the ways that Swift programmers will be able to write high performance GPU code in plain Swift. Chris Lattner discusses kernel fusion, XLA, and MLIR, which are exciting technologies coming soon to Swift programmers.

Then Jeremy talks about something that's available right now: amazingly great C interop. He shows how to use this to quickly and easily get high performance code by interfacing with existing C libraries, using Sox audio processing, and VIPS and OpenCV image processing as complete working examples.

C Header Files

```
// excerpt from math.h
#define M_E  2.71828182845904523536
extern double sqrt(double);
void __sincos(double x,
              double *sinp, double *cosp);


// excerpt from stdlib.h -> _malloc.h
void *malloc(size_t size);
void free(void *);
void *realloc(void *ptr, size_t size);
```

Generated Swift Interface

```
// excerpt from generated interface to math.h
var M_E: Double { get } /* e */
func sqrt(_: Double) -> Double
func __sincos(_ x: Double,
              _ sinp: UnsafeMutablePointer<Double>!,
              _ cosp: UnsafeMutablePointer<Double>!)


// excerpt from generated interface to _malloc.h
func malloc(_ size: Int) -> UnsafeMutableRawPointer!
func free(_: UnsafeMutableRawPointer!)
func realloc(_ ptr: UnsafeMutableRawPointer, _ size:Int)
       -> UnsafeMutableRawPointer!
```

Behind the scenes of Swift's C interop

Next up, we implement the Data Block API in Swift! Well… actually in some ways it's even *better* than the original Python version. We take advantage of an enormously powerful Swift feature: *protocols* (aka *type classes*).

```
let il = ItemList(fromFolder: path, extensions: ["jpeg", "jpg"])
let sd = SplitData(il) {grandParentSplitter(fName: $0, valid: "val")}
var procLabel = CategoryProcessor()
let sld = makeLabeledData(sd, fromFunc: parentLabeler, procLabel: &procLabel)
let rawData = sld.toDataBunch(itemToTensor: pathsToTensor, labelToTensor: intsToTensor, bs: 128)
let data = transformData(rawData) { openAndResize(fname: $0, size: 128) }
let batch = data.train.oneBatch()!
```

```
let labels = batch.yb.scalars.map { procLabel.vocab![Int($0)] }
showImages(batch.xb, labels: labels)
```

Data blocks API in Swift!

We now have enough Swift knowledge to implement a complete fully connect network forward pass in Swift—so that's what we do! Then we start looking at the backward pass, and use Swift's optional *reference semantics* to replicate the PyTorch approach. But then we learn how to do the same thing in a more "Swifty" way, using *value semantics* to do the backward pass in a really concise and flexible manner.

Finally, we put it all together, implementing our generic optimizer, Learner, callbacks, etc, to train Imagenette from scratch! The final notebooks in Swift show how to build and use much of the fastai.vision library in Swift, even although in these two lessons there wasn't time to cover everything. So be sure to study the notebooks to see lots more Swift tricks…

Further information

- Skip the FFI: Embedding Clang for C Interoperability
- Value Semantics talk by @AlexisGallagher
- Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions

More lessons

We'll be releasing even more lessons in the coming months and adding them to an attached course we'll be calling *Applications of Deep Learning*. They'll be linked from the Part 2 course page, so keep an eye out there. The first in this series will be a lesson about audio processing and audio models. I can't wait to share it with you all!