CSCI 5922 - NEURAL NETWORKS AND DEEP LEARNING

DEEP LEARNING SOFTWARE

HISTORY OF FRAMEWORKS

FIRST GENERATION

Framework	Year
Torch	2002
Theano	2010
Torch7	2011
Theano	2012
Pylearn2	2013
Keras	2015
Torchnet	2015

First frameworks to support pure automatic differentiation.

SECOND GENERATION

Framework	Year
Autograd	2015
Chainer	2015
MXNet	2015
TensorFlow	2016
Theano	2016
DyNet	2017
PyTorch	2017
Ignite	2018
JAX	2018
TensorFlow	2019

Wrappers

First full frameworks to support CUDA/GPUs.

2015 - DEEP LEARNING REDISCOVERS PURE AUTOMATIC DIFFERENTIATION

ALL OF THE FRAMEWORKS DISCUSSED SO FAR REQUIRE THE PROGRAMMER TO DEFINE THE COMPUTATIONAL GRAPH PRIOR TO RUNNING IT AND THE **GRAPH** IS **STATIC** (IN CHAINER NOMENCLATURE, THESE FRAMEWORKS ARE **DEFINE-THEN-RUN**).

THE **STATIC GRAPH**, **DEFINE-THEN-RUN** FRAMEWORKS LIMIT THE PROGRAMMER'S ABILITY TO EXPRESS COMPUTATIONS USING THE PROGRAMMING LANGUAGES NATIVE CONSTRUCTS, SUCH AS LOOPS AND CONDITIONALS.

STARTING IN 2015, MANY NEW FRAMEWORKS STARTED TO USE **DYNAMIC GRAPHS** (IN CHAINER NOMENCLATURE, THESE ARE **DEFINE-BY-RUN**).

THIS PARADIGM IS ESSENTIALLY THE SAME AS AUTOMATIC DIFFERENTIATION BY METHOD OVERLOADING, WHICH WE SAW IN AUTOMATIC DIFFERENTIATION IN MACHINE LEARNING: A SURVEY AND WHICH WAS NOT INVENTED BY THE DEEP LEARNING COMMUNITY

LOGISTIC REGRESSION IN TWO SLIDES

AUTOGRAD'S NUMPY WRAPS NUMPY

LIKE THEANO, A FUNCTION FOR COMPUTING GRADIENT OF LOSS WITH RESPECT TO PARAMETERS

```python
import autograd.numpy as np
from autograd import grad
from autograd.test_util import check_grads
```

STANDARD LOGISTIC REGRESSION STUFF

```python
def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according
    # to logistic model.
    return sigmoid(np.dot(inputs, weights))
```

THE LOSS, WHICH GETS WRAPPED BY AUTOGRAD

```python
def training_loss(weights, inputs, targets):
    # Training loss is the negative log-likelihood of the
    # training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + \
        (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))
```

LOGISTIC REGRESSION IN TWO SLIDES

INITIALIZATION OF DATA, WEIGHTS

`GRAD` RETURNS A FUNCTION THAT WRAPS ALL CONTINUOUS, DIFFERENTIABLE TRANSFORMATIONS

MODES CAN BE 'FWD' OR 'REV'

HOW DOES GRADIENT FUNCTION KNOW THE PARAMETERS OF THE MODEL?

```
# Build a toy dataset and weights.
inputs = np.array([[0.52, 1.12,  0.77],
                   [0.88, -1.08, 0.15],
                   [0.52, 0.06, -1.30],
                   [0.74, -2.49, 1.39]])
targets = np.array([True, True, False, True])
weights = np.array([0.0, 0.0, 0.0])

# Build a function that returns gradients of training loss
# with respect to parameters.
training_gradient_fun = grad(training_loss)

# Check the gradients numerically, just to be safe.
check_grads(training_loss, modes=['rev'])(
    weights, inputs, targets)

# Optimize weights using gradient descent.
print("Initial loss:", training_loss(weights, inputs, targets))
for i in range(10):
    weights -= training_gradient_fun(
        weights, inputs, targets) * 0.01
print("Trained loss:", training_loss(weights, inputs, targets))
```

- Chainer
  - First deep learning framework with both
    - Pure AD framework, dynamic graph, define-by-run
    - GPU support
  - CuPy
    - Low-level numerical library used by Chainer
    - Near drop-in replacement for NumPy
    - Just for GPUs

- Lots of support for production deployment (e.g. TensorFlow Serving, TensorFlow.js)
- *Fairly* straightforward to build preprocessing into the computational graph
  - This is super helpful for reproducibility and production use cases. Why?
- Static computational graph
  - Allows graph to be optimized prior to execution (in principle)
    - In practice, see XLA (Accelerated Linear Algebra), which is just-in-time.
  - When flow control is required, developer needs to become fluent in new API (e.g. `tf.cond`, `tf.while_loop`)
  - When print statements are required, the developer needs to use an API
    - `x = tf.Print(x, data=[x.size()], message='Length of vector')`
- Wanton use of Python context managers (e.g. with `tf.variable_scope(…)`)

DYNET - 2017

STATIC GRAPHS (THEANO, TENSORFLOW)

1. DEFINE GRAPH
2. FOR EACH DATA POINT
   I. ADD DATA
   II. FORWARD
   III. BACKWARD
   IV. UPDATE

- GRAPH CAN BE IMMEDIATELY OPTIMIZED WHEN DEFINED
- EASY TO SERIALIZE
- HARD TO IMPLEMENT VARYING STRUCTURE
- ERRORS ARE DEFERRED

DYNAMIC GRAPHS+EAGER EVALUATION (CHAINER, PYTORCH)

1. FOR EACH DATA POINT
   I. DEFINE/ADD DATA/FORWARD
   II. BACKWARD
   III. UPDATE

- GRAPH CANNOT BE IMMEDIATELY OPTIMIZED WHEN DEFINED
- HARDER TO SERIALIZE
- EASY TO IMPLEMENT VARYING STRUCTURE
- ERRORS ARE IMMEDIATE

DYNAMIC GRAPHS+LAZY EVALUATION (DYNET)

1. FOR EACH DATA POINT
   I. DEFINE/ADD DATA
   II. FORWARD
   III. BACKWARD
   IV. UPDATE

- GRAPH CANNOT BE IMMEDIATELY OPTIMIZED WHEN DEFINED
- HARDER TO SERIALIZE
- EASY TO IMPLEMENT VARYING STRUCTURE
- ERRORS ARE DEFERRED

Simple and Efficient Learning with Automatic Operation Batching, Neubig

On-the-fly Operation Batching in Dynamic Computation Graphs, Neubig, Goldberg, and Dyer, 2017

PURE AD PYTHON IMPLEMENTATION OF TORCH7, INSPIRED BY AUTOGRAD AND CHAINER.

LINEAR REGRESSION IN ONE SLIDE

CONTAINERS, SUCH AS SEQUENTIAL, INHERITED FROM TORCH7.

```
import torch

batch_size, input_dim, hidden_dim, output_dim = 64, 1000, 100, 10

x = torch.randn(batch_size, input_dim)
y = torch.randn(batch_size, output_dim)

model = torch.nn.Sequential(
    torch.nn.Linear(input_dim, hidden_dim),
    torch.nn.ReLU(),
    torch.nn.Linear(hidden_dim, output_dim))

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-4

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    print(t, loss.item())
    model.zero_grad()
    loss.backward()
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

CLEAR GRADIENTS FROM PREVIOUS ITERATION (ELSE THEY ARE ADDED).

NEW GRADIENTS COMPUTED HERE.

PYTORCH - 2017

DEFINING A SUBCLASS OF NN.MODULE CREATES YOUR OWN CONTAINER

STATELESS, FUNCTIONAL API, INSPIRED BY CHAINER. OBECT-ORIENTED API IS A FRONT-END OF THE FUNCTIONAL API.

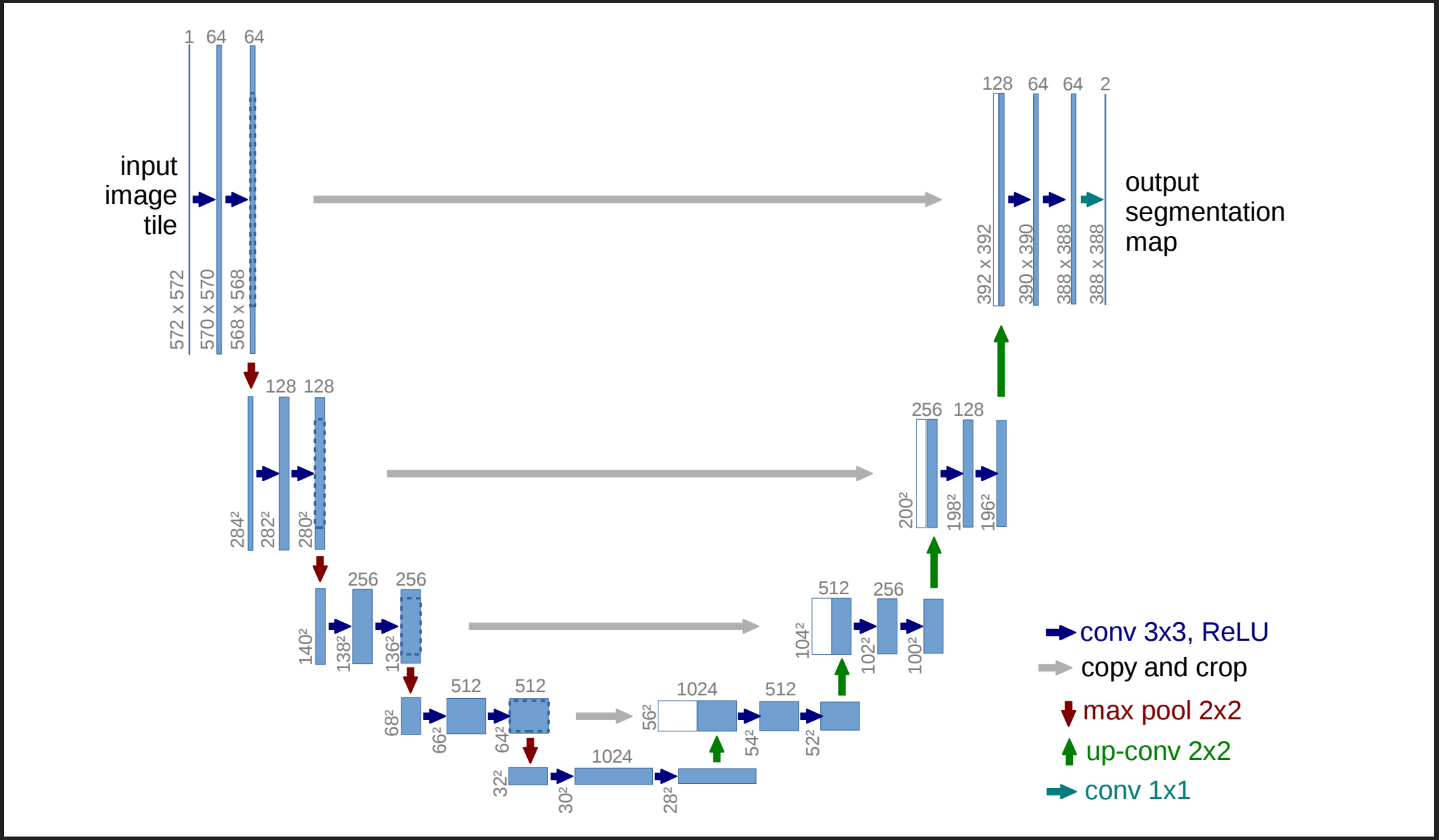
NN.MODULE IS THE PARENT OF PYTORCH'S CONTAINERS

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Network(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        # When defining your own subclass of Module, always call the
        # superclass' initializer before assigning to `self`. Doing
        # so ensures that parameter updates occur for all properties
        # of instances of this subclass.
        super().__init__()

        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, input):
        output = self.linear1(input)
        output = F.relu(output)
        output = self.linear2(output)
        return output
```

ONLY NEED TO DEFINE FORWARD. NN.MODULE HANDLES THE BACKWARD PASS FOR YOU.

LOOPS, IFS, RECURSION — YOU NAME IT — CAN HAPPEN IN FORWARD.

A SIMPLE UNET IN TWO SLIDES.



WE WON'T REPRODUCE ITS NAMESAKE EXACTLY.

A SIMPLE UNET IN TWO SLIDES.

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class UNetModule(nn.Module):
    def __init__(self, in_channels, out_channels, mode=None):
        super().__init__()
        assert mode in ['encoder', 'decoder']
        self.mode = mode

        if self.mode is 'encoder':
            self.conv1 = nn.Conv2d(in_channels, out_channels, 3,
                                   stride=2, padding=1, bias=False)
        else:
            self.conv1 = nn.Conv2d(in_channels, out_channels, 3,
                                   stride=1, padding=1, bias=False)
        self.act1 = nn.ReLU(out_channels)

        if self.mode == 'decoder':
            self.transposed = nn.ConvTranspose2d(
                out_channels, out_channels, 3,
                stride=2, padding=1, output_padding=1, bias=False)

    def forward(self, input, lateral_input=None):
        if self.mode == 'decoder' and lateral_input is not None:
            input = torch.cat((input, lateral_input), dim=1)
        output = self.conv1(input)
        output = self.act1(output)
        if self.mode == 'decoder':
            output = self.transposed(output)
        return output
```

PYTORCH - 2017

A SIMPLE UNET IN TWO SLIDES.

```
class UNet(nn.Module):
    def __init__(self, n_classes, n_input_channels=3):
        super().__init__()

        self.n_classes = n_classes

        self.encoder1 = UNetModule(n_input_channels, 16, mode='encoder')
        self.encoder2 = UNetModule(16, 16, mode='encoder')
        self.encoder3 = UNetModule(16, 16, mode='encoder')
        self.encoder4 = UNetModule(16, 16, mode='encoder')

        self.decoder4 = UNetModule(16, 16, mode='decoder')
        self.decoder3 = UNetModule(32, 16, mode='decoder')
        self.decoder2 = UNetModule(32, 16, mode='decoder')
        self.decoder1 = UNetModule(32, 16, mode='decoder')

        self.classifier = nn.Conv2d(16, n_classes, 1, padding=0, bias=False)

    def forward(self, input):
        encoder1_output = self.encoder1(input)
        encoder2_output = self.encoder2(encoder1_output)
        encoder3_output = self.encoder3(encoder2_output)
        encoder4_output = self.encoder4(encoder3_output)

        decoder4_output = self.decoder4(encoder4_output)
        decoder3_output = self.decoder3(decoder4_output, encoder3_output)
        decoder2_output = self.decoder2(decoder3_output, encoder2_output)
        decoder1_output = self.decoder1(decoder2_output, encoder1_output)

        output = self.classifier(decoder1_output)

        return output
```

RECALL THAT WHEN A TENSOR TAKES MULTIPLE PATHS THROUGH A NETWORK (I.E. THERE'S A FORK IN THE ROAD, AND IT TAKES BOTH), THERE ARE — DURING THE BACKWARD PASS — MULTIPLE GRADIENTS, ONE FOR EACH PATH THE TENSOR TOOK.

ALL FRAMEWORKS SUM THESE GRADIENTS BY DEFAULT. GETTING ACCESS TO THE PRE-SUMMED GRADIENTS CAN BE TRICKY.

THE NEXT THREE SLIDES SHOW ONE WAY TO DO THIS.

PYTORCH SUPPORTS HOOKS ON BOTH MODULES AND TENSORS. DEFINE FUNCTIONS TO PRINT THE GRADIENTS.

```python
#!/usr/bin/env python

# coding: utf-8

from functools import partial

import torch
import torch.nn as nn


def print_tensor_grad(grad, name=None, value=None):
    print(name, 'value', value, 'grad', grad)

def print_module_grad(module, grad_input, grad_out, name=None):
    print(name, grad_input)
```

NOW DEFINE THE NETWORK ITSELF. THE NETWORK MUST INHERIT FROM TORCH.NN.MODULE AND MUST CALL THE SUPERCLASS'S INITIALIZER BEFORE ASSIGNING TO SELF IN ITS OWN INITIALIZER. WHY?

A MODULE IS A CONTAINER. FOR IT TO KNOW WHICH OF ITS PROPERTIES ARE PARAMETERS, THE SUPERCLASS OVERRIDES __SETATTR__, SO WHEN YOU WRITE SELF.LAYER1 = NN.LINEAR(…), IT CAN REGISTER SELF.LAYER1 AS A CHILD MODULE AND AUTOMATICALLY UPDATE ITS PARAMETERS DURING TRAINING.

```
class Network(nn.Module):
    def __init__(self, n_in=2, n_out=2):
        super().__init__()
        self.layer1 = nn.Linear(n_in, n_out, bias=False)
        self.layer2 = nn.Linear(n_out, n_out, bias=False)
        self.layer3 = nn.Linear(n_out*2, 1, bias=False)
        self.fun = nn.LeakyReLU(negative_slope=1.0)
```

TO GET ACCESS TO THE PRE-SUMMED GRADIENTS, WE'LL ADD A LEAKY RELU WITH A NEGATIVE SLOPE OF 1 TO THE COMPUTATIONAL GRAPH, AND ADD A HOOK TO IT. WHY?

PYTORCH - 2017 - GRADIENT HACKING

```
class Network(nn.Module):
    # Initializer skipped here. See previous slide.

    def forward(self, input):
        out1 = self.layer1(input)
        out1.retain_grad()

        path1 = self.fun(out1)
        path1.retain_grad()
        path2 = self.fun(out1)
        path2.retain_grad()

        out2 = self.layer2(path1)
        input3 = torch.cat((path2, out2), dim=1)
        out3 = self.layer3(input3)

        out1.register_hook(
            partial(print_tensor_grad, name='out1', value=out1))
        path1.register_hook(
            partial(print_tensor_grad, name='path1', value=path1))
        path2.register_hook(
            partial(print_tensor_grad, name='path2', value=path2))

        return {
            'out1': out1,
            'path1': path1,
            'path2': path2,
            'y': out3
        }
```

THE RETAIN_GRAD METHOD INSTRUCTS PYTORCH'S TORCH.AUTOGRAD MODULE TO KEEP THE GRADIENTS FOR A NODE IN THE GRAPH AFTER THE REVERSE PASS. TO PRESERVE MEMORY, IT CLEARS UNNEEDED GRADIENTS.

WHAT IS GOING ON HERE? HOW DOES THIS ALLOW US TO GET ACCESS TO THE PRE-SUMMED GRADIENTS OF LAYER1?

NOW RUN IT!

```
if __name__ == '__main__':
    torch.manual_seed(17)
    network = Network()
    x = torch.ones(1, 2)
    out = network(x)
    out['y'].backward()
    # Verify that the gradient of the output of the first layer is the
    # same as the sum of the two paths taken by that output.
    print('out1', out['out1'].grad)
    print('path1', out['path1'].grad)
    print('path2', out['path2'].grad)
    assert torch.all(
        out['out1'].grad == out['path1'].grad + out['path2'].grad)
```

PYTORCH IGNITE - 2018

THESE KEEP THE SIMPLE CASE SIMPLE WITH A DEFAULT ENGINE.

EVENT HANDLERS CAN BE ADDED VIA A DECORATOR.

```python
# Instantiate the optimizer, trainer, and evaluator.
optimizer = SGD(model.parameters(), lr=lr, momentum=momentum)
trainer = create_supervised_trainer(
        model, optimizer, F.nll_loss, device=device)
evaluator = create_supervised_evaluator(
        model, device=device,
        metrics={'accuracy': Accuracy(), 'nll': Loss(F.nll_loss)})

desc = 'ITERATION - loss: {:.2f}'
pbar = tqdm(
    initial=0, leave=False, total=len(train_loader), desc=desc.format(0))

@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(engine):
    # When the model has seen every example in the training set, run the
    # validation set and report metrics.
    evaluator.run(val_loader)
    metrics = evaluator.state.metrics
    avg_accuracy = metrics['accuracy']
    avg_nll = metrics['nll']
    tqdm.write(
        'Val - Epoch: {}  Avg accuracy: {:.2f} Avg loss: {:.2f}'.format(
            engine.state.epoch, avg_accuracy, avg_nll))
    pbar.n = pbar.last_print_n = 0

# Now train the model.
trainer.run(train_loader, max_epochs=epochs)
pbar.close()
```

DEFINING YOUR OWN PROCESS FUNCTION.

```
def train_and_store_loss(engine, batch):
    # The process function gets called on each iteration.
    inputs, targets = batch
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
    return loss.item()

engine = Engine(train_and_store_loss)

@engine.on(Events.COMPLETED)
def cleanup(engine):
    print('Done training on epoch {:04d}'.format(engine.state.epoch))

# Can also add handlers via a method.
engine.add_event_handler(Events.COMPLETED, cleanup)

engine.run(data_loader)
```

EVENTS IN IGNITE

- Events.STARTED
- Events.COMPLETED
- Events.EPOCH_STARTED
- Events.EPOCH_COMPLETED
- Events.ITERATION_STARTED
- Events.ITERATION_COMPLETED
- Events.EXCEPTION_RAISED

- NumPy drop-in replacement with just-in-time compilation
- CPU or GPU
- Automatic vectorization
- Uses TensorFlow's XLA backend to compile quickly for GPU

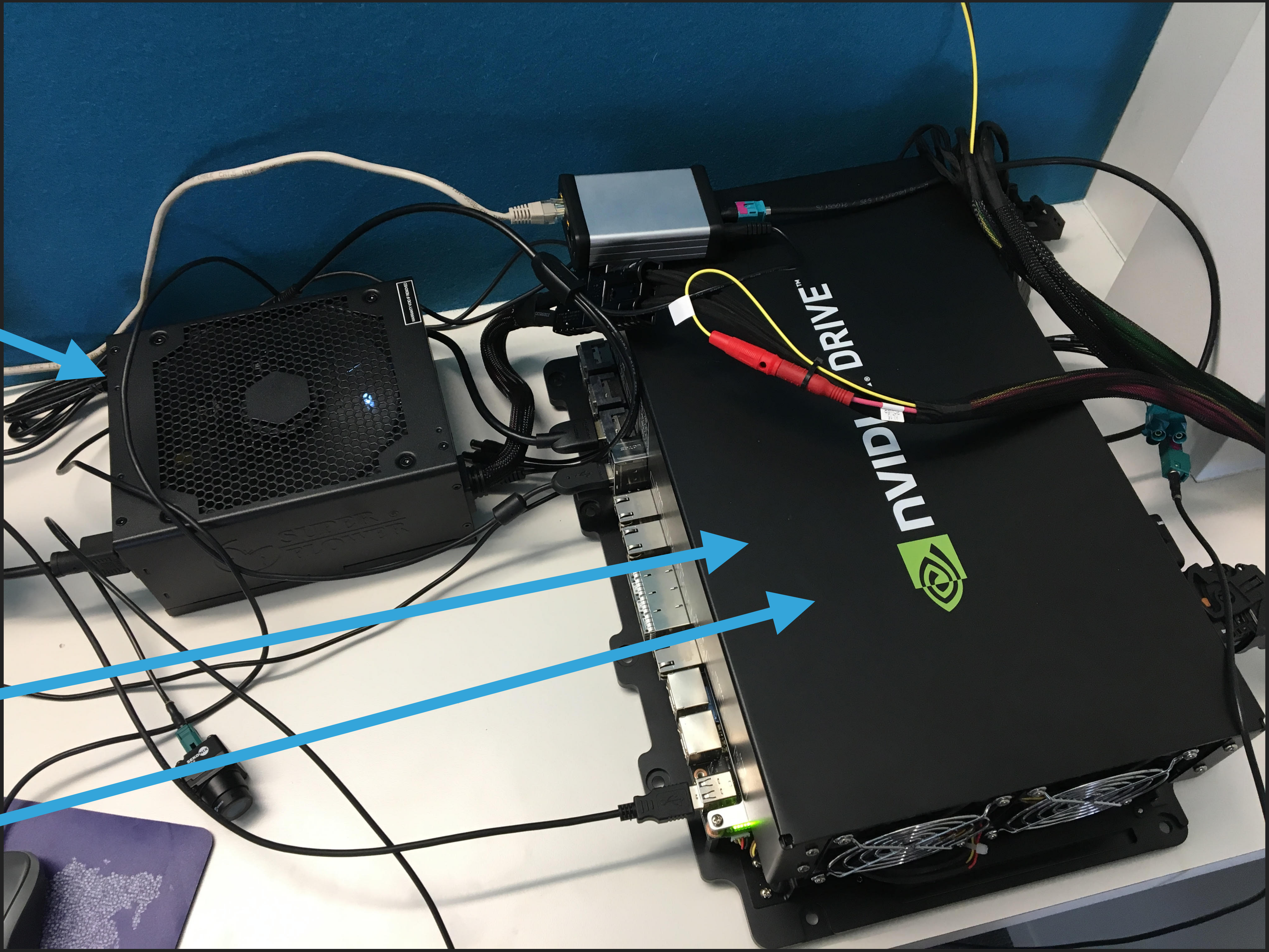
- Super important changes
  - Eager mode will be the default
    - Graphs will be dynamic
    - Stack traces will still be ugly
  - Keras will be the default way of using the API
    - TensorFlow will look more like Chainer and PyTorch

- Mobile deployment environments
  - Mobile GPUs
    - NVIDIA Jetson TX1, TX2, Xavier AGX
      - Low-power envelope, single GPU, shared memory devices for resource-constrained deployments
      - Jetson TX* have shared memory between CPU and GPU

CONVOLUTIONAL NETWORKS ARE MOVING TO THE EDGE

NVIDIA DRIVE PEGASUS IS DESIGNED FOR ROBOTAXIS.

500 WATT POWER SUPPLY



16 ARM64 CPUS

2 VOLTA GPUS

- Mobile deployment environments
  - FPGAs - see talk by Phil James-Roxby this semester
- Mobile phones
  - iOS: Metal Performance Shaders
  - Android and iOS: TensorFlow Lite
  - ONNX (also for general interchange/optimization)

- Allows programming of convnets on platforms that support Apple's Metal API
  - Swift and Objective-C (iOS), possibly C++ on Mac OS
  - Support for constructing multi-layer neural networks, as well as convolutional or recurrent ones
  - Some support for training, but appeal is inference - significant speed-ups reported using MPS

TENSORFLOW LITE

- TensorFlow support for GPUs on
  - iOS devices is based on Metal Performance Shaders
  - Android devices is based on OpenGL Compute Shaders
- The GPU Delegate
  - Prunes unnecessary operations
  - Replaces some operations with faster versions
  - Fuses some sequences of operations (e.g. fusing batch norm affine weights into convolution)
  - Falls back to the CPU for operations that are not implemented for the GPU
- Pre-trained models available
  - MobileNet v1 for image classification, PoseNet for pose estimation, DeepLab segmentation, MobileNet SSD object detection