torch-workshop

RMACC 2023 - Arizona State University

Nathan Stephens

Welcome

About Me

- Currently Senior Manager for Developer Relations at NVIDIA.
- Background in analytic solutions and consulting.
- Experience building data science teams, architecting analytic infrastructure, and operationalizing data products.
- Worked at RStudio, oversaw the solutions engineering team.
- Longtime advocate for operationalizing data science using open-source software.
- Relocated to Peoria, AZ in 2020.

Deep Learning and Scientific Computing With R Torch

This workshop presents a conceptual overview of the R interface for PyTorch. Emphasis will be on basic structures and applications for deep learning. Materials are based on the new book, Deep Learning and Scientific Computing with R torch by Sigrid Keydana (2023).

About the book

This is a book about `torch`, the R interface to PyTorch. PyTorch, as of this writing, is one of the major deep-learning and scientific-computing frameworks, widely used across industries and areas of research.

1. Introduction to core `torch`: Basic structures, automatic differentiation, optimizers, etc.
2. Applications of deep learning: Image recognition, time series, audio classification, etc.
3. Non-deep-learning: Matrix computations, calculating the Discrete Fourier Transform, and wavelet analysis.

Workshop Contents

Covering first two sections:

- Getting familiar with torch
  - Torch mechanics: `3`, 4, 9
  - What can we do? 5, `6`, 10
  - Abstraction: 7, 8, `11`
- Deep learning with torch
  - Basic image classification: 13, `14`, `15`
  - Opitmized image classification: 16, 17, 18

Remaining chapters are left to the reader

Setup

CPU Setup (Windows and Mac)

1. Install R and RStudio
2. Install `torch`

```
install.packages("torch")
```

3. Clone the torch-workshop repos and make sure the working directory is `torch-workshop`

```
git clone https://github.com/nwstephens/torch-workshop
```

4. Download `Tiny ImageNet` data

```
tiny_imagenet_dataset(".", download = TRUE)
```

5. Install the `modeldata` package from the `2021-06-08` snapshot

```
install.packages("modeldata", repos = "https://packagemanager.rstudio.com/cran/__linux__/b
```

GPU Accelerated R Setup

- Obtaining an NVIDIA GPU
  - Rig
  - Cloud
  - Data Center
- Installing and configuring an NVIDIA GPU
  - CUDA Toolkit
  - NVIDIA Container Toolkit
- Installing and configuring R
  - The Rocker Project
  - Posit Package Manager
  - R Binaries

GPU Setup (Linux)

1. Have a CUDA compatible NVIDIA GPU with compute capability 6.0 or higher
2. Install the NVIDIA Container Toolkit
3. Pull and run the RStudio Rocker container

```
docker pull rocker/rstudio
docker run --gpus=all -d -t -e PASSWORD=rstudio -p 8787:8787 --name rstudio rocker/rstudio
```

4. Open RStudio Server from your browser by opening `http://<your.ip.address>:8787/`. Make sure port 8787 is open. Your username is `rstudio` and your password is `rstudio`.
5. Configure RStudio Server to download package binaries from the Posit Package Manager

```
# For Red Hat 8 use:
https://packagemanager.rstudio.com/cran/__linux__/centos8/latest

# For Ubuntu 20.04 use:
https://packagemanager.rstudio.com/cran/__linux__/focal/latest
```

6. Install `torch` from the pre-built binaries (Warning! This download is 2Gb)

```
options(timeout = 600)
install.packages("torch", repos = "https://storage.googleapis.com/torch-lantern-builds/pac
```

7. Make sure your CUDA device is available (this should return `TRUE`)

```
library(torch)
cuda_is_available()
```

8. Clone this repos and make sure the working directory is `torch-workshop`
9. Download `Tiny ImageNet` data

```
wget http://cs231n.stanford.edu/tiny-imagenet-200.zip
unzip tiny-imagenet-200.zip
```

10. Install the `modeldata` package from the `2021-06-08` snapshot

```
install.packages("modeldata", repos = "https://packagemanager.rstudio.com/cran/__linux__/b
```

Getting Familiar with torch

2. What is torch?

`torch` is an R port of PyTorch.

It is written entirely in R and C++ (including a bit of C). No Python installation is required to use it.

There's a vibrant community of developers

The `torch for R` ecosystem also includes R packages: `torchvision` , `torchaudio`, and `luz`.

3. Tensors [Exercise]

What's in a tensor?

To do anything useful with `torch`, you need to know about tensors. Not tensors in the math/physics sense. In deep learning frameworks such as TensorFlow and (Py-)Torch, *tensors* are "just" multi-dimensional arrays optimized for fast computation – not on the CPU only but also, on specialized devices such as GPUs and TPUs.

In fact, a `torch tensor` is like an R `array`, in that it can be of arbitrary dimensionality. But unlike `array`, it is designed for fast and scalable execution of mathematical calculations, and you can move it to the GPU. (It also has an extra capability of enormous practical impact – automatic differentiation – but we reserve that for the next chapter.)

Topics:

1. Creating tensors

2. Operations on tensors
3. Accessing parts of a tensor
4. Reshaping tensors
5. Broadcasting

4. Autograd

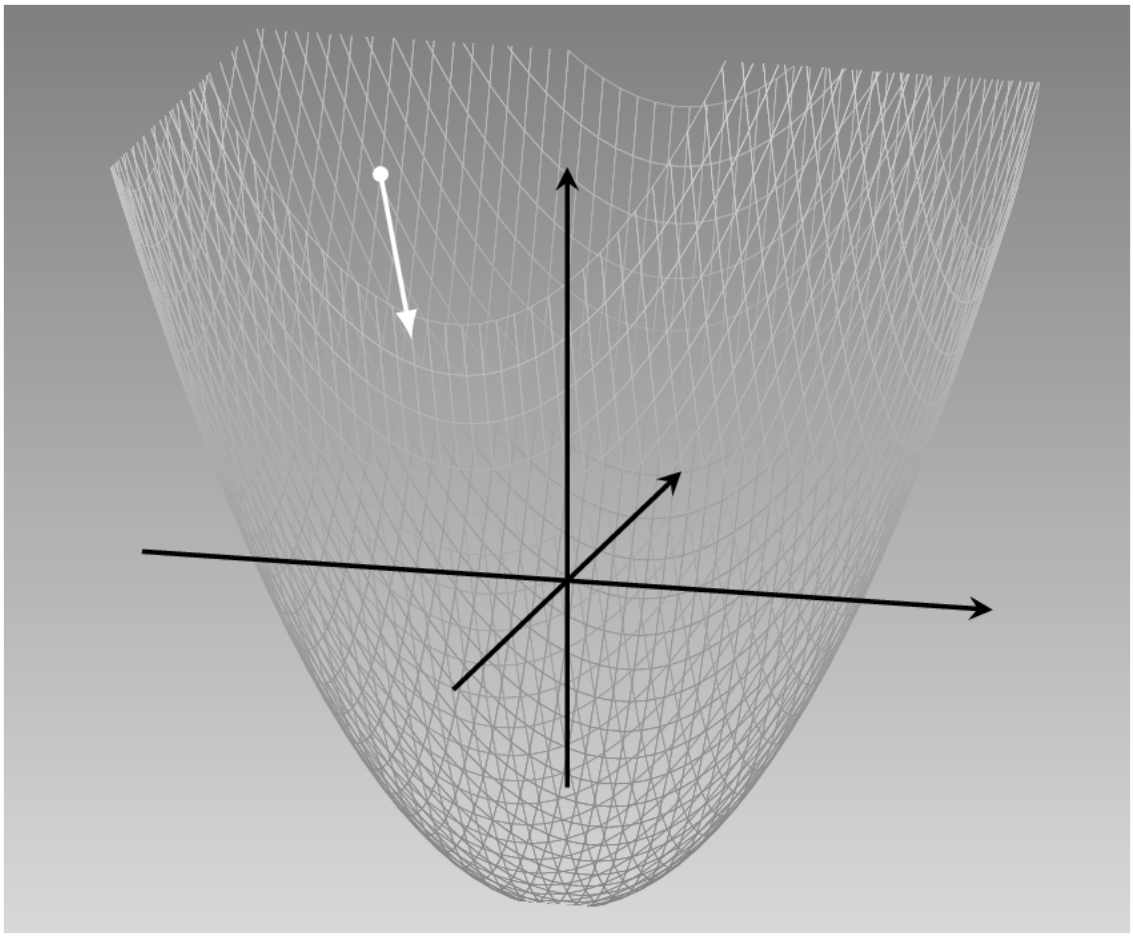
Frameworks like `torch` are so popular because of what you can do with them: deep learning, machine learning, optimization, large-scale scientific computation in general. Most of these application areas involve minimizing some *loss function*. This, in turn, entails computing function *derivatives*.

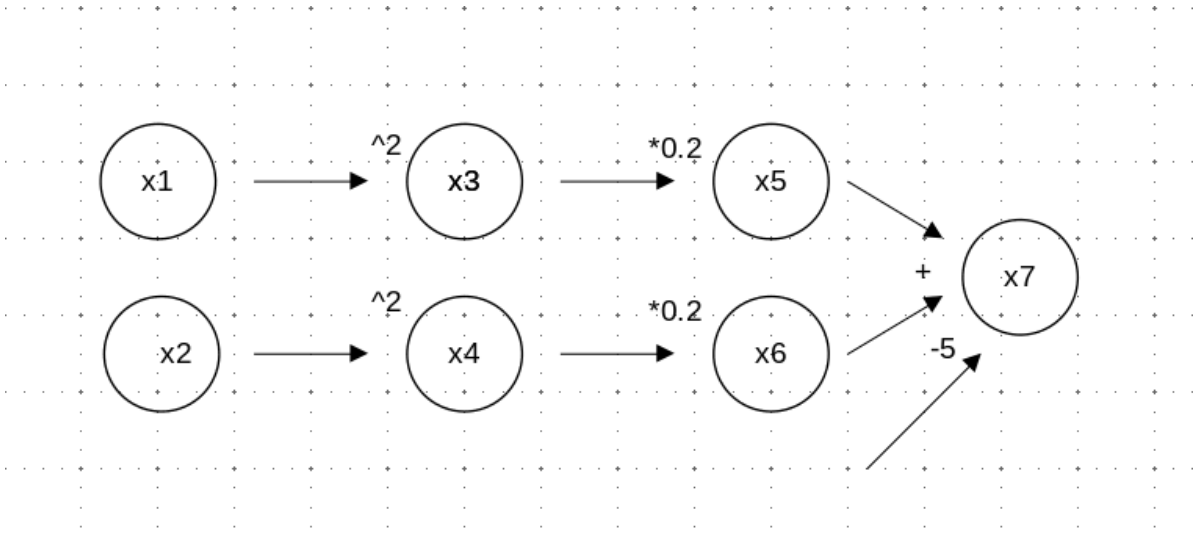
`torch` implements what is called *automatic differentiation*. In automatic differentiation, and more specifically, its often-used *reverse-mode* variant, derivatives are computed and combined on a *backward pass* through the graph of tensor operations.

In torch, the AD engine is usually referred to as autograd.

Automatic differentiation example

Quadratic function of two variables: $f(x_1, x_2) = 0.2{x_1}^2 + 0.2{x_2}^2 - 5$. It has its minimum at `(0,0)`.




x1
^2
x3
*0.2
x5
x2
^2
x4
*0.2
x6
+
x7
-5

- At `x7`, we calculate partial derivatives with respect to `x5` and `x6`. Basically, the equation to differentiate looks like this: $f(x_5, x_6) = x_5 + x_6 - 5$. Thus, both partial derivatives are 1.
- From `x5`, we move to the left to see how it depends on `x3`. We find that $\frac{\partial x_5}{\partial x_3} = 0.2$. At this point, applying the chain rule of calculus, we already know how the output depends on `x3`: $\frac{\partial f}{\partial x_3} = 0.2 * 1 = 0.2$.
- From `x3`, we take the final step to `x`. We learn that $\frac{\partial x_3}{\partial x_1} = 2x_1$. Now, we again apply the chain rule, and are able to formulate how the function depends on its first input: $\frac{\partial f}{\partial x_1} = 2x_1 * 0.2 * 1 = 0.4x_1$.
- Analogously, we determine the second partial derivative, and thus, already have the gradient available: $\nabla f = \frac{\partial f}{\partial x_1} + \frac{\partial f}{\partial x_2} = 0.4x_1 + 0.4x_2$.

Automatic differentiation with `torch` autograd

```
library(torch)

x1 <- torch_tensor(2, requires_grad = TRUE)
x2 <- torch_tensor(2, requires_grad = TRUE)

x3 <- x1$square()
x5 <- x3 * 0.2

x4 <- x2$square()
x6 <- x4 * 0.2

x7 <- x5 + x6 - 5
x7
```

```
torch_tensor
-3.4000
[ CPUFloatType{1} ][ grad_fn = <SubBackward1> ]
```

```
x7$backward()

x1$grad
x2$grad
```

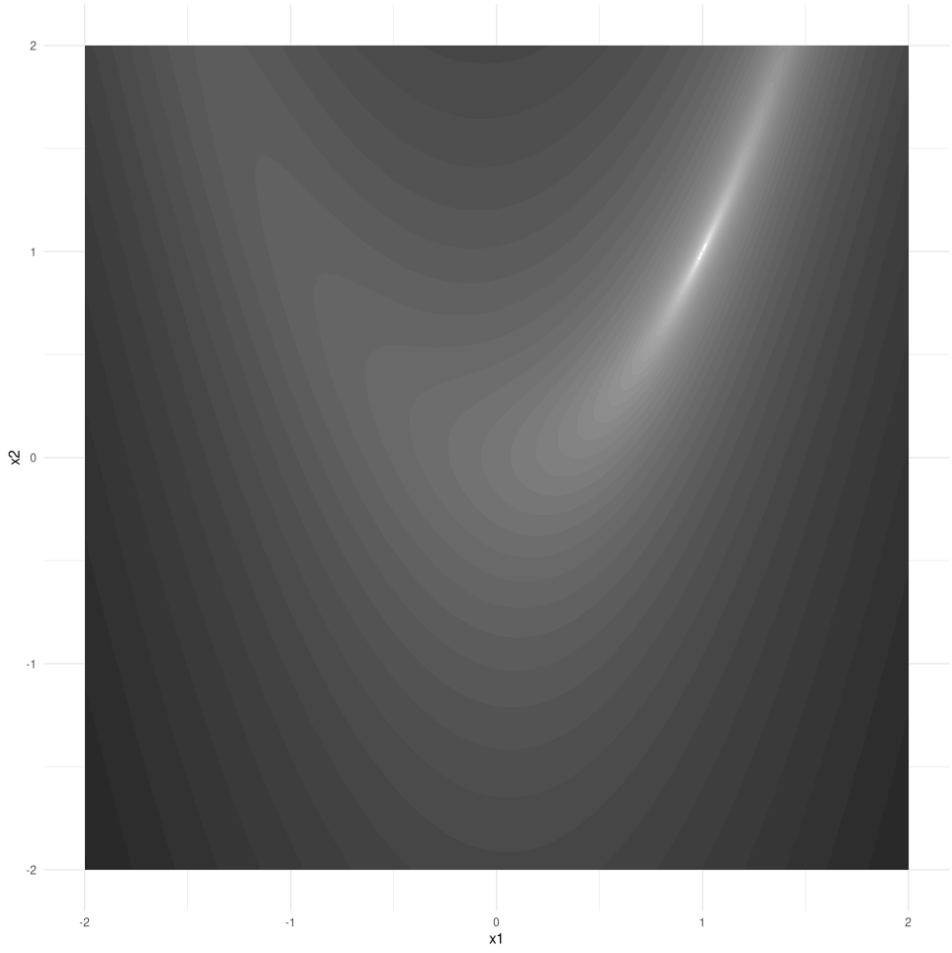
```
 0.8000
[ CPUFloatType{1} ]
```

```
torch_tensor
 0.8000
[ CPUFloatType{1} ]
```

These are the partial derivatives of `x7` with respect to `x1` and `x2`, respectively. Conforming to our manual calculations above, both amount to 0.8, that is, 0.4 times the tensor values 2 and 2.

5. Function minimization with *autograd*

In optimization research, the *Rosenbrock function* is a classic. It is a function of two variables; its minimum is at `(1,1)`. If you take a look at its contours, you see that the minimum lies inside a stretched-out, narrow valley.



```
a <- 1
b <- 5

rosenbrock <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  (a - x1)^2 + b * (x2 - x1^2)^2
}
```

- `lr`, for learning rate, is the fraction of the gradient to subtract on every step
- `num_iterations` is the number of steps to take.

```
library(torch)

num_iterations <- 1000

lr <- 0.01

x <- torch_tensor(c(-1, 1), requires_grad = TRUE)

for (i in 1:num_iterations) {
  if (i %% 100 == 0) cat("Iteration: ", i, "\n")

  value <- rosenbrock(x)
  if (i %% 100 == 0) {
    cat("Value is: ", as.numeric(value), "\n")
  }

  value$backward()
  if (i %% 100 == 0) {
    cat("Gradient is: ", as.matrix(x$grad), "\n")
  }

  with_no_grad({
    x$sub_(lr * x$grad)
    x$grad$zero_()
  })
}
```

```
Iteration:  400
Value is:  0.009619333
Gradient is:  -0.04347242 -0.08254051
```

```
Iteration:  500
Value is:  0.003990697
Gradient is:  -0.02652063 -0.05206227

Iteration:  600
Value is:  0.001719962
Gradient is:  -0.01683905 -0.03373682

Iteration:  700
Value is:  0.0007584976
Gradient is:  -0.01095017 -0.02221584

Iteration:  800
Value is:  0.0003393509
Gradient is:  -0.007221781 -0.01477957

Iteration:  900
Value is:  0.0001532408
Gradient is:  -0.004811743 -0.009894371

Iteration:  1000
Value is:  6.962555e-05
Gradient is:  -0.003222887 -0.006653666
```

After thousand iterations, we have reached a function value lower than 0.0001. What is the corresponding `(x1,x2)`-position?

```
x
```

```
torch_tensor
 0.9918
 0.9830
[ CPUFloatType{2} ]
```

This is rather close to the true minimum of `(1,1)`.

6. A neural network from scratch [Exercise]

Understanding the basics will be an efficient antidote against the surprisingly common temptation to think of deep learning as some kind of "magic". It's all just matrix computations; one has to learn how to orchestrate them though.

Multiple Linear Regression

$$f(\mathbf{X}) = \mathbf{X}\mathbf{W} + \mathbf{b}$$

```
library(torch)

x <- torch_randn(100, 3)

w <- torch_randn(3, 1, requires_grad = TRUE)

b <- torch_zeros(1, 1, requires_grad = TRUE)

y <- x$matmul(w) + b

print(y, n = 10)
```

```
torch_tensor
-2.1600
-3.3244
 0.6046
 0.4472
-0.4971
-0.0530
 5.1259
-1.1595
-0.5960
-1.4584
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{100,1} ][ grad_fn = <AddBackward0> ]
```

Layers

$$g(f(\mathbf{X}))$$

$$g \circ f$$

One of the defining features of neural networks is their ability to chain an unlimited (in theory) number of layers.

All but the output layer may be referred to as "hidden" layers, although from the point of view of someone who uses a deep learning framework such as `torch`, they are not that *hidden* after all.

```
# Hidden Layer
w1 <- torch_randn(3, 8, requires_grad = TRUE)
b1 <- torch_zeros(1, 8, requires_grad = TRUE)

# Output Layer
w2 <- torch_randn(8, 1, requires_grad = TRUE)
b2 <- torch_randn(1, 1, requires_grad = TRUE)
```

$$\begin{aligned} f(\mathbf{X}) &= (\mathbf{X}\mathbf{W}_1)\mathbf{W}_2 \\ &= \mathbf{X}(\mathbf{W}_1\mathbf{W}_2) \\ &= \mathbf{X}\mathbf{W}_3 \end{aligned}$$

Activation functions

The most-used activation function inside a network is the so-called *ReLU*, or Rectified Linear Unit. This is a long name for something rather straightforward: All negative values are set to zero. In `torch`, this can be accomplished using the `relu()` function:

```
t <- torch_tensor(c(-2, 1, 5, -7))
t$relu()
```

```
torch_tensor
 0
 1
 5
 0
[ CPUFloatType{4} ]
```

Loss functions

Loss is a measure of how far away we are from our goal. For regression-type tasks, this often will be mean squared error (MSE).

```
y <- torch_randn(5)
y_pred <- y + 0.01
```

```
loss <- (y_pred - y)$pow(2)$mean()

loss
```

```
torch_tensor
9.99999e-05
[ CPUFloatType{} ]
```

Example

- do a forward pass, yielding the network's predictions (if you dislike the one-liner, feel free to split it up);
- compute the loss (this, too, being a one-liner – we merely added some logging);
- have *autograd* calculate the gradient of the loss with respect to the parameters; and
- update the parameters accordingly (again, taking care to wrap the whole action in `with_no_grad()`, and zeroing the `grad` fields on every iteration).

Generate random data

```
library(torch)

## input dimensionality (number of input features)
d_in <- 3
## number of observations in training set
n <- 100

x <- torch_randn(n, d_in)
coefs <- c(0.2, -1.3, -0.5)
y <- x$matmul(coefs)$unsqueeze(2) + torch_randn(n, 1)
```

Build the network

```
# dimensionality of hidden layer
d_hidden <- 32
# output dimensionality (number of predicted features)
d_out <- 1

# weights connecting input to hidden layer
w1 <- torch_randn(d_in, d_hidden, requires_grad = TRUE)
# weights connecting hidden to output layer
w2 <- torch_randn(d_hidden, d_out, requires_grad = TRUE)
```

```
# hidden layer bias
b1 <- torch_zeros(1, d_hidden, requires_grad = TRUE)
# output layer bias
b2 <- torch_zeros(1, d_out, requires_grad = TRUE)
```

Train the network

```
learning_rate <- 1e-4

### training loop --------------------------------------
for (t in 1:200) {

  ### -------- Forward pass --------
  y_pred <- x$mm(w1)$add(b1)$relu()$mm(w2)$add(b2)

  ### -------- Compute loss --------
  loss <- (y_pred - y)$pow(2)$mean()
  if (t %% 10 == 0)
    cat("Epoch: ", t, "   Loss: ", loss$item(), "\n")

  ### -------- Backpropagation --------
  # compute gradient of loss w.r.t. all tensors with
  # requires_grad = TRUE
  loss$backward()

  ### -------- Update weights --------
  # Wrap in with_no_grad() because this is a part we don't
  # want to record for automatic gradient computation
   with_no_grad({
     w1 <- w1$sub_(learning_rate * w1$grad)
     w2 <- w2$sub_(learning_rate * w2$grad)
     b1 <- b1$sub_(learning_rate * b1$grad)
     b2 <- b2$sub_(learning_rate * b2$grad)

     # Zero gradients after every pass, as they'd
     # accumulate otherwise
     w1$grad$zero_()
     w2$grad$zero_()
     b1$grad$zero_()
     b2$grad$zero_()
   })
```

```
}
```

```
Epoch: 10 Loss: 24.92771
Epoch: 20 Loss: 23.56143
Epoch: 30 Loss: 22.3069
Epoch: 40 Loss: 21.14102
Epoch: 50 Loss: 20.05027
Epoch: 60 Loss: 19.02925
Epoch: 70 Loss: 18.07328
Epoch: 80 Loss: 17.16819
Epoch: 90 Loss: 16.31367
Epoch: 100 Loss: 15.51261
Epoch: 110 Loss: 14.76012
Epoch: 120 Loss: 14.05348
Epoch: 130 Loss: 13.38944
Epoch: 140 Loss: 12.77219
Epoch: 150 Loss: 12.19302
Epoch: 160 Loss: 11.64823
Epoch: 170 Loss: 11.13535
Epoch: 180 Loss: 10.65219
Epoch: 190 Loss: 10.19666
Epoch: 200 Loss: 9.766989
```

7. Modules

In `torch`, a module can be of any complexity, ranging from basic *layers* to complete *models* consisting of many such layers. Code-wise, there is no difference between "layers" and "models".

`nn_module()`

In `torch`, a linear layer is created using `nn_linear()` which expects (at least) two arguments: `in_features` and `out_features`.

```
library(torch)
l <- nn_linear(in_features = 5, out_features = 16)
l
```

```
An `nn_module` containing 96 parameters.
Parameters
 weight: Float [1:16, 1:5]
 bias: Float [1:16]
```

```
l$weight
```

```
torch_tensor
-0.2079 -0.1920  0.2926  0.0036 -0.0897
 0.3658  0.0076 -0.0671  0.3981 -0.4215
 0.2568  0.3648 -0.0374 -0.2778 -0.1662
 0.4444  0.3851 -0.1225  0.1678 -0.3443
-0.3998  0.0207 -0.0767  0.4323  0.1653
 0.3997  0.0647 -0.2823 -0.1639 -0.0225
 0.0479  0.0207 -0.3426 -0.1567  0.2830
 0.0925 -0.4324  0.0448 -0.0039  0.1531
-0.2924 -0.0009 -0.1841  0.2028  0.1586
-0.3064 -0.4006 -0.0553 -0.0067  0.2575
-0.0472  0.1238 -0.3583  0.4426 -0.0269
-0.0275 -0.0295 -0.2687  0.2236  0.3787
-0.2617 -0.2221  0.1503 -0.0627  0.1094
 0.0122  0.2041  0.4466  0.4112  0.4168
-0.4362 -0.3390  0.3679 -0.3045  0.1358
 0.2979  0.0023  0.0695 -0.1906 -0.1526
[ CPUFloatType{16,5} ]
```

```
l$bias
```

```
torch_tensor
-0.2314
 0.2942
 0.0567
-0.1728
-0.3220
-0.1553
-0.4149
-0.2103
-0.1769
 0.4219
-0.3368
 0.0689
 0.3625
-0.1391
-0.1411
-0.2014
[ CPUFloatType{16} ]
```

```
x <- torch_randn(50, 5)
output <- l(x)
output$size() # foward pass
```

```
[1] 50 16
```

`nn_sequential()`

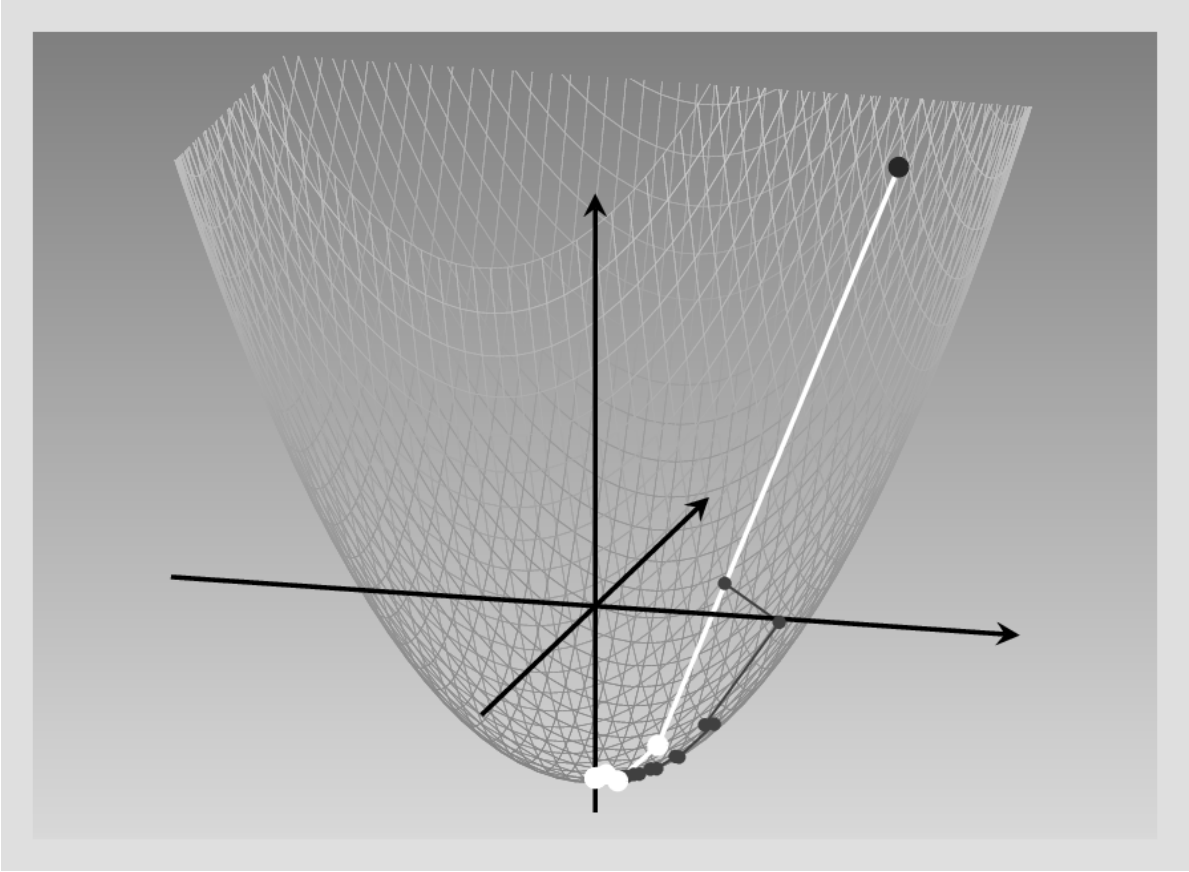
If all our model should do is propagate straight through the layers, we can use `nn_sequential()` to build it. Models consisting of all linear layers are known as *Multi-Layer Perceptrons* (MLPs).

```
mlp <- nn_sequential(
  nn_linear(10, 32),
  nn_relu(),
  nn_linear(32, 64),
  nn_relu(),
  nn_linear(64, 1)
)
```

Other modules

- `nn_conv1d()`, `nn_conv2d(), and nn_conv3d()`, the so-called *convolutional* layers that apply filters to input data of varying dimensionality,
- `nn_lstm()` and `nn_gru()` , the *recurrent* layers that carry through a state,
- `nn_embedding()` that is used to embed categorical data in high-dimensional space,
- and more.

8. Optimizers



1. Optimizers help in updating weights

Without optimizers

```
library(torch)

# compute gradient of loss w.r.t. all tensors with
# requires_grad = TRUE
loss$backward()

### -------- Update weights --------

# Wrap in with_no_grad() because this is a part we don't
# want to record for automatic gradient computation
with_no_grad({
```

```
  w1 <- w1$sub_(learning_rate * w1$grad)
  w2 <- w2$sub_(learning_rate * w2$grad)
  b1 <- b1$sub_(learning_rate * b1$grad)
  b2 <- b2$sub_(learning_rate * b2$grad)

  # Zero gradients after every pass, as they'd accumulate
  # otherwise
  w1$grad$zero_()
  w2$grad$zero_()
  b1$grad$zero_()
  b2$grad$zero_()
})
```

With optimizers

```
# compute gradient of loss w.r.t. all tensors with
# requires_grad = TRUE
# no change here
loss$backward()

# Still need to zero out gradients before the backward pass,
# only this time, on the optimizer object
optimizer$zero_grad()

# use the optimizer to update model parameters
optimizer$step()
```

2. Optimizers help minimize the loss function

Without optimizers

- Stochastic gradient descent (baseline)

With optimizers

- Gradient descent (with momentum)
- Adagrad (adaptive learning rate)
- RMSProp (adapative learning rate)
- Adam (adapative learning rate and momentum)

9. Loss functions

Mean squared error

```
library(torch)
loss <- (y_pred - y)$pow(2)$sum()
```

In `torch`, loss functions start with `nn_` or `nnf_`.

```
nnf_mse_loss(torch_ones(2, 2), torch_zeros(2, 2) + 0.1)
```

```
torch_tensor
0.81
[ CPUFloatType{} ]
```

What loss function should I choose?

	Data		**Input**		
	binary	multi-class	raw scores	probabilities	log probs
BCeL	Y		Y		
Ce		Y	Y		
BCe	Y			Y	
Nll		Y			Y

BCeL	`nnf_binary_cross_entropy_with_logits()`
Ce	`nnf_cross_entropy()`
BCe	`nnf_binary_cross_entropy()`
Nll	`nnf_nll_loss()`

10. Function minimization with L-BFGS

So far, we've only talked about the kinds of optimizers often used in deep learning – stochastic gradient descent (SGD), SGD with momentum, and a few classics from the *adaptive learning rate* family: RMSProp, Adadelta, Adagrad, Adam. All these have in common one thing: They only make use of the *gradient*, that is, the vector of first derivatives. Accordingly, they are all *first-order* algorithms. This means, however, that they are missing out on helpful information provided by the *Hessian*, the matrix of second derivatives.

Approximate Newton: BFGS and L-BFGS

Among approximate Newton methods, probably the most-used is the Broyden-Goldfarb-Fletcher-Shanno algorithm, or BFGS. Instead of continually computing the exact inverse of the Hessian, it keeps an iteratively-updated approximation of that inverse. BFGS is often implemented in a more memory-friendly version, referred to as Limited-Memory BFGS (L-BFGS). This is the one provided as part of the core torch optimizers.

Line search

With line search, we spend some time evaluating how far to follow the descent direction. There are two principal ways of doing this.

1. Exact. Take the current point, compute the descent direction, and hard-code them as givens in a *second* function that depends on the learning rate only. Then, differentiate this function to find *its* minimum. The solution will be the learning rate that optimizes the step length taken.
2. Approximate search. Essentially, we look for something that is *just good enough*. Among the most established heuristics are the *Strong Wolfe conditions*, and this is the strategy implemented in `torch`'s `optim_lbfgs()`.

11. Modularizing the neural network [Exercise]

- The forward pass, instead of calling functions on tensors, will call the model.
- In computing the loss, we now make use of `torch`'s `nnf_mse_loss()`.
- Backpropagation of gradients is, in fact, the only operation that remains unchanged.
- Weight updating is taken care of by the optimizer.

Training

```
opt <- optim_adam(net$parameters)

#### training loop --------------------------------------

for (t in 1:200) {

  ### -------- Forward pass --------
  y_pred <- net(x)

  ### -------- Compute loss --------
  loss <- nnf_mse_loss(y_pred, y)
  if (t %% 10 == 0)
    cat("Epoch: ", t, "   Loss: ", loss$item(), "\n")

  ### -------- Backpropagation --------
```

```
  opt$zero_grad()
  loss$backward()

  ### -------- Update weights --------
  opt$step()

}
```

Deep learning with torch

12. Overview

The upcoming two chapters will introduce you to workflow-related techniques that are indispensable in practice. You'll encounter another package, `luz`, that endows `torch` with an important layer of abstraction, and significantly streamlines the workflow. Once you know how to use it, we're all set to look at a first application: image classification.

Regarding workflow, we'll see how to:

- prepare the input data in a form the model can work with;
- effectively and efficiently train a model, monitoring progress and adjusting hyperparameters on the fly;
- save and load models;
- making models generalize beyond the training data;
- speed up training;
- and more.

13. Loading data

Using `dataset()`

Overview

```
ds <- dataset()(
  initialize = function(...) {
    ...
  },
  .getitem = function(index) {
    ...
  },
```

```
  .length = function() {
    ...
  }
)
```

Example

```
library(torch)
library(palmerpenguins)
library(dplyr)

penguins_dataset <- dataset(
  name = "penguins_dataset()",
  initialize = function(df) {
    df <- na.omit(df)
    self$x <- as.matrix(df[, 3:6]) %>% torch_tensor()
    self$y <- torch_tensor(
      as.numeric(df$species)
    )$to(torch_long())
  },
  .getitem = function(i) {
    list(x = self$x[i, ], y = self$y[i])
  },
  .length = function() {
    dim(self$x)[1]
  }
)

ds[1]
```

```
$x
torch_tensor
   39.1000
   18.7000
  181.0000
 3750.0000
[ CPUFloatType{4} ]

$y
torch_tensor
1
[ CPULongType{} ]
```

Using `tensor_dataset()`

```
penguins <- na.omit(penguins)
ds <- tensor_dataset(
  torch_tensor(as.matrix(penguins[, 3:6])),
  torch_tensor(
    as.numeric(penguins$species)
  )$to(torch_long())
)
```

Using `dataloader()`

```
dl <- dataloader(ds, batch_size = 32, shuffle = TRUE)

first_batch <- dl %>%
  # obtain an iterator for this dataloader
  dataloader_make_iter() %>%
  dataloader_next()

dim(first_batch$x)
dim(first_batch$y)
```

```
[1] 32  1 28 28
[1] 32
```

14. Training with luz [Exercise]

`luz` was designed to make deep learning with `torch` as effortless as possible. `luz` helps manage:

- Training, validation, and testing
- Customization flexibility with callbacks
- Data flow between *devices* (CPU and GPU, if you have one).

```
fitted <- net %>%
  setup(
    loss = nn_mse_loss(),
    optimizer = optim_adam,
    metrics = list(luz_metric_mae())
  ) %>%
  set_hparams(d_in = d_in,
```

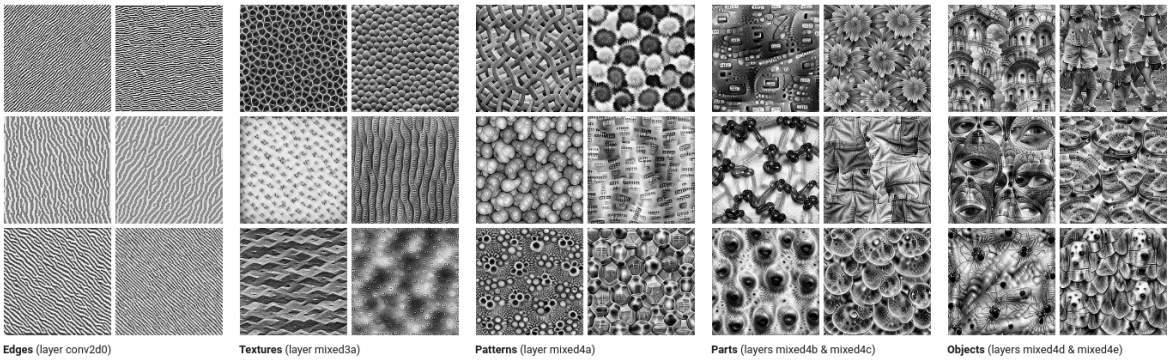
```
              d_hidden = d_hidden,
              d_out = d_out) %>%
  fit(
    train_dl,
    epochs = 200,
    valid_data = valid_dl,
    callbacks = list(
      luz_callback_model_checkpoint(path = "./models/",
                                    save_best_only = TRUE),
      luz_callback_early_stopping(patience = 10)
    )
  )

fitted %>% predict(test_dl)
```

How `luz` helps with devices. For the call to `predict()`, what happened "under the hood" was the following:

- `luz` put the model in evaluation mode, making sure that weights are not updated.
- `luz` moved the test data to the GPU, batch by batch, and obtained model predictions.
- These predictions were then moved back to the CPU, in anticipation of the caller wanting to process them further with R.

15. A first go at image classification [Exercise]

This chapter is about "convolutional" neural networks; the specialized module in question is the "convolutional" one. In this chapter, we design and train a basic convnet from scratch. This is "just" a beginning.



Glossary of terms

- Convolution / Cross-correlation: Which finds things, or spots similarities.

- Filter / kernel: Maps non-linear data into a higher-dimensional space without the need to visit or understand that higher-dimensional space.
  - Padding: Allows the kernel to extend outside the valid region.
  - Stride: The way a kernel moves over the image.
  - Dilation: Spreads out the pixels in a convolution.
- Translation invariant: If a shift has occurred, the target is still detected, but at a new location.
- Pooling: Down sample feature maps by summarizing the presence of features in patches of the feature map.

New Package

In addition to `torch` and `luz`, we load a third package from the `torch` ecosystem: `torchvision`. `torchvision` provides operations on images, as well as a set of pre-trained models and common benchmark datasets.

Dataset

```
library(torch)
library(torchvision)
library(luz)

set.seed(777)
torch_manual_seed(777)

train_ds <- tiny_imagenet_dataset(
  root=".",
  download = FALSE,
  transform = function(x) {
    x %>%
      transform_to_tensor()
  }
)

valid_ds <- tiny_imagenet_dataset(
  root=".",
  split = "val",
  transform = function(x) {
    x %>%
      transform_to_tensor()
  }
)
```

Dataloader

```
train_dl <- dataloader(train_ds,
  batch_size = 128,
  shuffle = TRUE
)
valid_dl <- dataloader(valid_ds, batch_size = 128)

batch <- train_dl %>%
  dataloader_make_iter() %>%
  dataloader_next()
```

Convolutional Neural Network (CNN or convnet)

```
convnet <- nn_module(
  "convnet",
  initialize = function() {
    self$features <- nn_sequential(
      nn_conv2d(3, 64, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_conv2d(64, 128, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_conv2d(128, 256, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_conv2d(256, 512, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_max_pool2d(kernel_size = 2),
      nn_conv2d(512, 1024, kernel_size = 3, padding = 1),
      nn_relu(),
      nn_adaptive_avg_pool2d(c(1, 1))
    )
    self$classifier <- nn_sequential(
      nn_linear(1024, 1024),
      nn_relu(),
      nn_linear(1024, 1024),
      nn_relu(),
      nn_linear(1024, 200)
    )
  },
```

```
  forward = function(x) {
    x <- self$features(x)$squeeze()
    x <- self$classifier(x)
    x
  }
)
```

Training

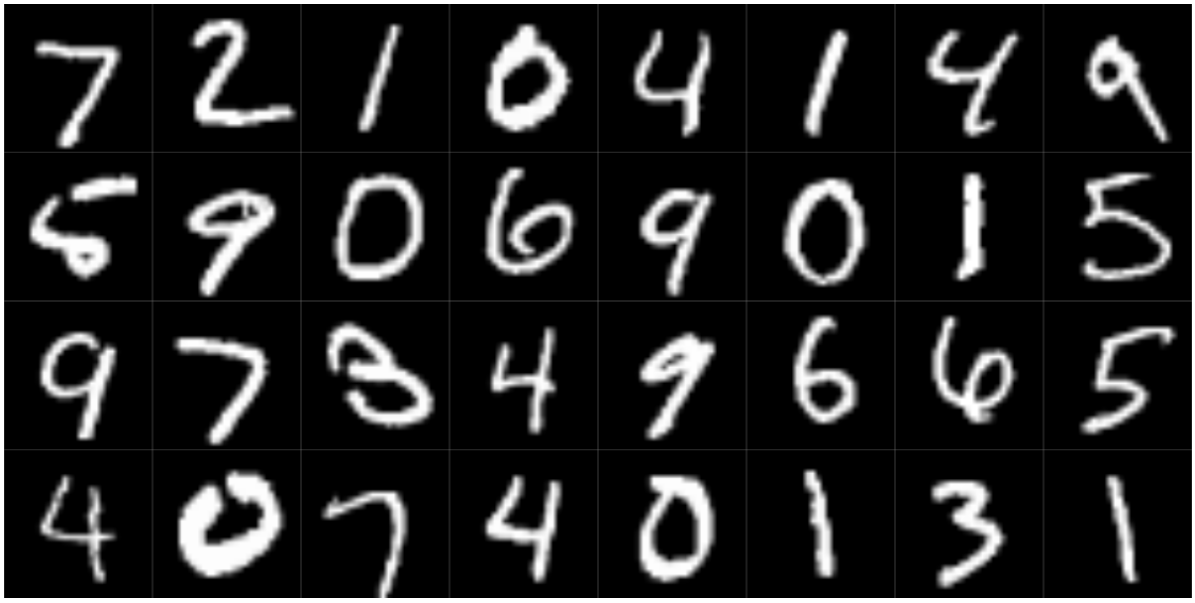
```
fitted <- convnet %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_adam,
    metrics = list(
      luz_metric_accuracy()
    )
  ) %>%
  fit(train_dl,
      epochs = 50,
      valid_data = valid_dl,
      verbose = TRUE
      )
```

Predicting

```
preds <- fitted %>% predict(valid_dl)

preds <- nnf_softmax(preds, dim = 2)

torch_argmax(preds, dim = 2)
```

16. Making models generalize

Original image



Rotations, flips, translations

Mixup

Dropout

At each forward pass, individual activations – single values in the tensors being passed on – are dropped (meaning: set to zero), with configurable probability. Put differently: Dynamically and reversibly, individual inter-neuron connections are "cut off".

Regularization

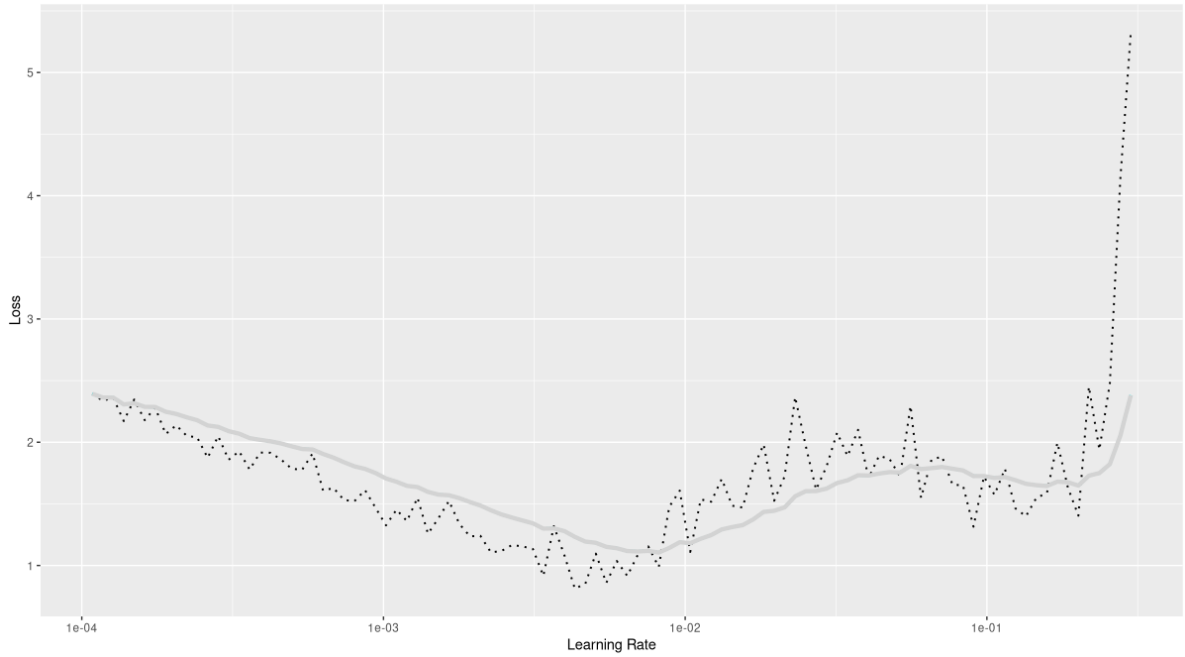
The idea is to keep the weights small and homogeneous, to prevent sharp cliffs and canyons in the loss function. It is often referred to as "weight decay". Regularization is not seen that often in the context of neural networks.

Early stopping

In deep learning, early stopping is ubiquitous; it's hard to imagine why one would not want to use it.

17. Speeding up training

1. Batch normalization. *Batchnorm* – to introduce a popular abbreviation – layers are added to a model to stabilize and, in consequence, speed up training.
2. Determining a good learning rate upfront, and dynamically varying it during training. As you might remember from our experiments with optimization, the learning rate has an enormous impact on training speed and stability.
3. Transfer learning. Applied to neural networks, the term commonly refers to using pre-trained models for feature detection, and making use of those features in a downstream task.



18.Image classification: Improving performance

Always use

- Data augmentation. There is hardly ever a case where you'd *not* want to use it – unless, of course, you are already using a different data augmentation technique.
- Learning rate finder with a learning rate schedule.
- Early stopping. This will not just prevent overfitting, but also, save time.

Exercises (slow)

- First training: Take the convnet from three chapters ago, and add dropout layers.
- Second training: Replace dropout by batch normalization. (Everything else stays the same.)
- Third training: Replace the model completely, by one chaining a pre-trained feature classifier (ResNet) and a small sequential model.

Appendix

Chapters left to the reader

- 19. Image segmentation

- 20. Tabular data
- 21. Time series
- 22. Audio classification
- 23. Overview
- 24. Matrix computations: Least-squares problems
- 25. Matrix computations: Convolution
- 26. Exploring the Discrete Fourier Transform (DFT)
- 27. The Fast Fourier Transform (FFT)
- 28. Wavelets

Thank you

- Nathan Stephens
- nstephens@nvidia.com

- torch-workshop