Neural Networks with torch & luz

Lecture 27

Dr. Colin Rundel

The abstraction ladder

We have walked up several layers of abstraction for fitting the same linear regression:

- Hand-coded gradient descent with an analytically derived gradient (`grad_desc_lm()`)
- `torch` tensors + autograd - we no longer need to derive the gradient by hand
- `nn_module` + `optim_sgd` - we no longer need to write the parameter update step
- `nn_linear` - we no longer need to manage the parameter tensors directly
- `luz` - we no longer need to write the training loop, batching, or metric tracking

Each layer trades a bit of explicitness for a lot of convenience (but less flexibility)

luz

What is `luz`?

`luz` is a high-level training API built on top of `torch`. It removes most of the training-loop boilerplate we wrote in the previous lecture:

- Automatic batching via dataloaders
- Built-in metric tracking, validation splits, callbacks
- Early stopping, learning rate schedulers, checkpoints
- A built-in loss-curve plot

luz is roughly the R analogue of PyTorch Lightning - the same `nn_module` we built by hand can be fit with luz with no

A quick recap

Recall the 10000-row regression problem we fit by hand last lecture using `nn_linear` and a manual training loop:

```
1 Xt$shape
```

```
[1] 10000    20
```

```
1 yt$shape
```

```
[1] 10000     1
```

```
1 coef(lm_fit) |> round(2)
```

```
(Intercept)          X1          X2          X3          X4
       3.03        0.01        0.00        5.22       -0.01
         X5          X6          X7          X8          X9
       0.00        0.00       -3.10       -0.01        0.00
        X10         X11         X12         X13         X14
      -0.01       -0.01        8.69       -0.02       -0.01
        X15         X16         X17         X18         X19
       0.02        0.02        0.01       -1.48        0.00
        X20
      -0.01
```

```
1 deviance(lm_fit) |> round(2)
```

```
[1] 9859.22
```

The luz workflow

```
fitted = nn_module(
  initialize = function(in_features) {
    self$linear = nn_linear(in_features, 1)
  },
  forward = function(X) {
    self$linear(X)
  }
) |>
  setup(
    loss = nn_mse_loss(),
    optimizer = optim_sgd
  ) |>
  set_hparams(in_features = p) |>
  set_opt_hparams(lr = 0.05) |>
  fit(
    list(Xt, yt),
    epochs = 20,
    dataloader_options = list(
      batch_size = 100,
      shuffle = TRUE
    ),
    verbose = FALSE
  )
```

The four-stage pipeline maps directly onto things we wrote by hand:

- `setup()` - pick a loss function and optimizer
- `set_hparams()` - arguments passed to the model's `initialize()`
- `set_opt_hparams()` - arguments passed to the optimizer
- `fit()` - run the training loop, with batching and shuffling handled for us

Inspecting the fit

```
1 c(fitted$model$linear$bias, fitted$model$linear$weight) |> map(as.numeric) |> unlist() |> rou
```

```
 [1]  2.98  0.02  0.00  5.26 -0.04  0.02  0.04 -3.09  0.01  0.02 -0.04
[12] -0.02  8.70 -0.02  0.02  0.02  0.03  0.00 -1.54 -0.02 -0.01
```

```
1 lm_fit |> coef() |> round(2) |> unname()
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01  0.00  0.00 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.69 -0.02 -0.01  0.02  0.02  0.01 -1.48  0.00 -0.01
```

```
1 fitted$records$metrics$train |> unlist() |> unname()
```

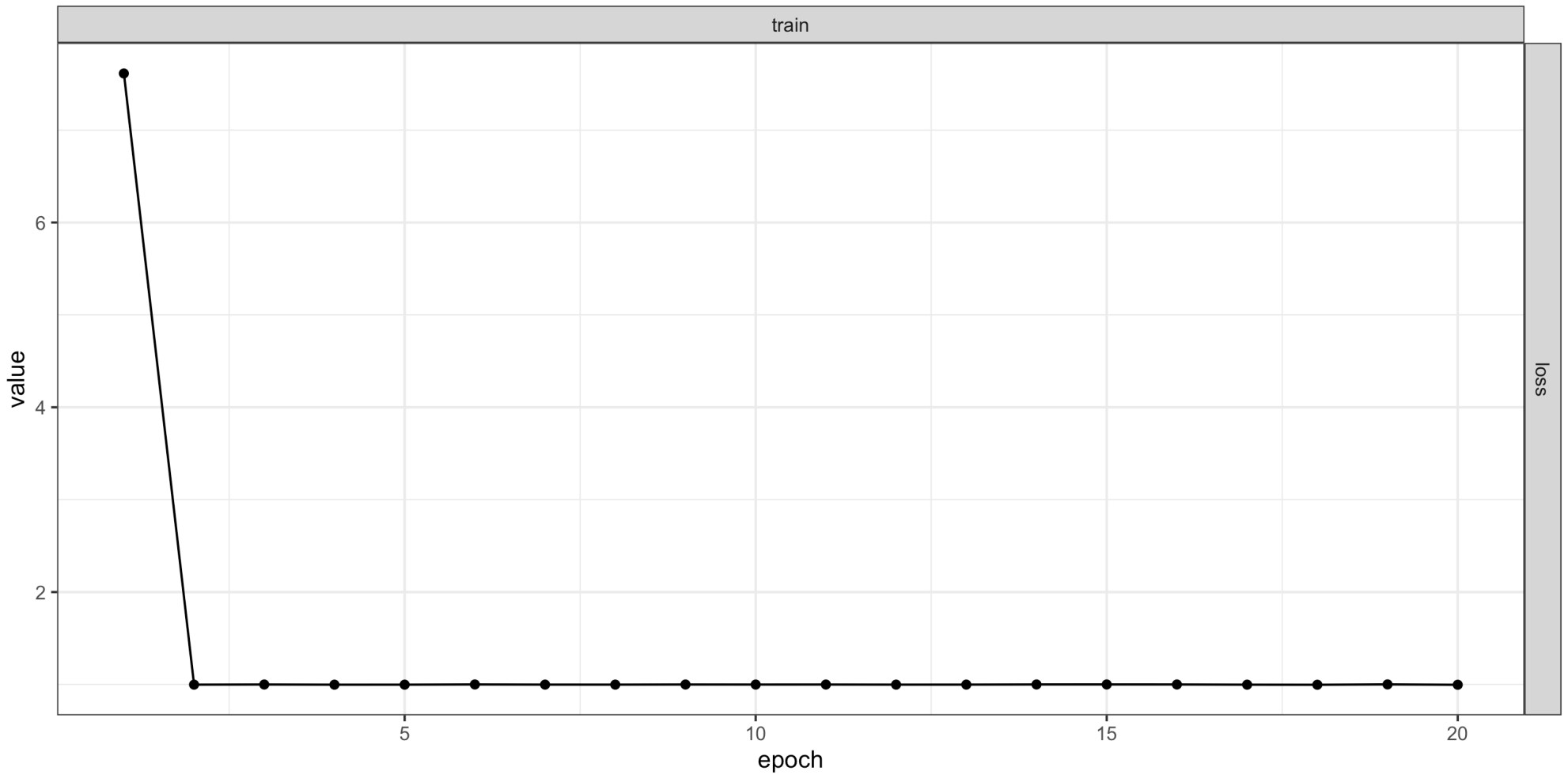
```
 [1] 7.6132669 0.9980766 0.9995083 0.9977403 0.9978724 1.0000237
 [7] 0.9983734 0.9982529 0.9994720 0.9989458 0.9992331 0.9981256
[13] 0.9984254 0.9998939 1.0000370 0.9997186 0.9977539 0.9969737
[19] 1.0005277 0.9968545
```

```
1 mean(lm_fit$residuals^2)
```

```
[1] 0.9859224
```

Loss curve from luz

```
1 plot(fitted)
```



MNIST

The digits dataset

We will now take a look at the MNIST dataset - this variant comes from Python sklearn package and is available as a CSV file in `static/slides/data/mnist_digits.csv`.

It contains 1797 8×8 grayscale images of handwritten digits.

```
digits = readr::read_csv("data/mnist_digits.csv", show_col_types = FALSE)
dim(digits)
```

```
[1] 1797   65
```

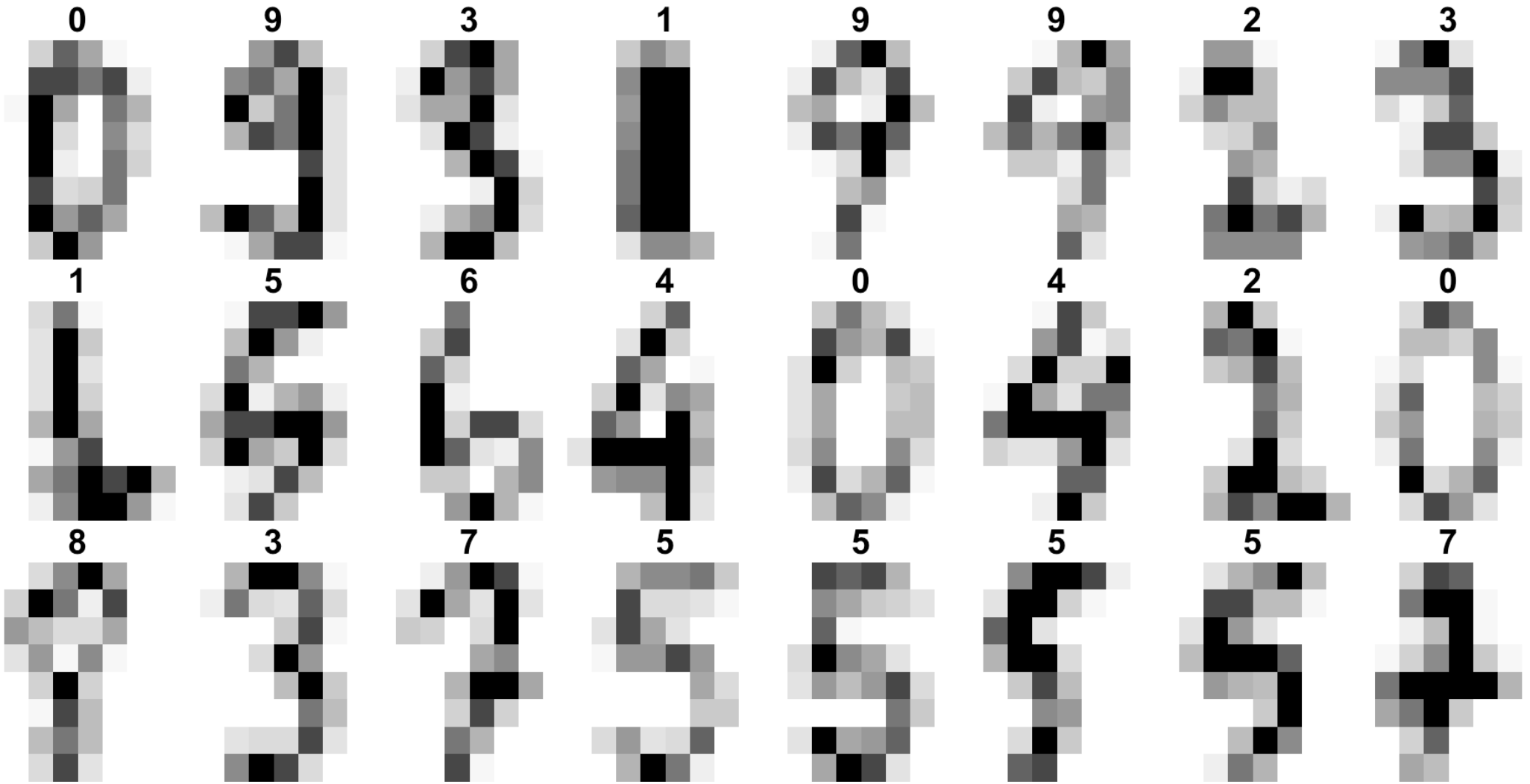
```
X = digits  |> select(-label) |> as.matrix()
y = digits$label
dim(X)
```

```
[1] 1797   64
```

```
table(y)
```

```
y
  0   1   2   3   4   5   6   7   8   9
178 182 177 183 181 182 181 179 174 180
```

Example digits



Train / test split

```
set.seed(1234)
split = initial_split(digits, prop = 0.8)

X_train = training(split) |> select(-label) |> as.matrix()
X_test  = testing(split)  |> select(-label) |> as.matrix()
y_train = training(split)$label
y_test  = testing(split)$label

dim(X_train); dim(X_test)
```

```
[1] 1437   64
```

```
[1] 360  64
```

To torch tensors

```
X_train_t = torch_tensor(X_train, dtype = torch_float())
y_train_t = torch_tensor(y_train + 1L, dtype = torch_long())
X_test_t  = torch_tensor(X_test,  dtype = torch_float())
y_test_t  = torch_tensor(y_test  + 1L, dtype = torch_long())
```

```
X_train_t$shape
```

```
[1] 1437   64
```

```
y_train_t$shape
```

```
[1] 1437
```

R torch's `nn_cross_entropy_loss()` expects **1-based** class indices (the R convention), so we add 1 to the 0-9 digit labels

`nn_linear`

A quick refresher - `nn_linear(in_features, out_features)` implements the affine map

$$y = xW^\top + b$$

with a learnable weight matrix `W` of shape `(out_features, in_features)` and bias vector of length `out_features`. This is just a slightly funny way of writing a traditional regression model.

- Input shape: `(N, in_features)` - one row per example
- Output shape: `(N, out_features)` - one column per output unit

In our Regression model we used `out_features = 1` since we were trying to predict a single number. For our classification model we will use `out_features = 10` - one per class - and interpret the output as raw class scores (logits).

A linear (multinomial) model

The following is the `torch` analogue of `nnet::multinom()` - a single `nn_linear` layer from 64 pixel features to 10 class scores, trained with cross-entropy loss.

```
mnist_linear = nn_module(
  initialize = function() {
    self$linear = nn_linear(64, 10)
  },
  forward = function(x) {
    self$linear(x)
  }
)
```

Cross-entropy loss

For K-class classification the standard loss is *cross-entropy*. Given raw class scores (logits) $z = (z_1, \dots, z_K)$ from the model, we convert to class probabilities with the softmax

$$p_k = \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)}$$

and the loss for a single example with true class y is the negative log-likelihood of the correct class

$$\ell(z, y) = -\log p_y = -z_y + \log \sum_{j=1}^{K} \exp(z_j).$$

Averaging over the mini-batch gives the objective that `nn_cross_entropy_loss()` minimizes.

`nn_cross_entropy_loss()`

A few things to keep in mind when using it:

- The model's `forward()` should return raw logits, not probabilities - the softmax is fused into the loss for numerical stability.
- Targets are class indices (a `long` tensor), not a dummy coded / one-hot encoded matrix
- R torch uses 1-based indexing, so digit labels 0-9 become 1-10 (this is why we added 1 earlier).

```
1 loss_fn = nn_cross_entropy_loss()
2 logits = torch_tensor(matrix(c(2.0, 0.5, -1.0, 0.1), nrow = 1))
3 target = torch_tensor(1L)
4 loss_fn(logits, target)
```

```
torch_tensor
0.352405935525894 17
[ CPUFloatType{} ]
```

```
1 -log(exp(2.0) / sum(exp(c(2.0, 0.5, -1.0, 0.1))))
```

```
[1] 0.3524059
```

Fitting with luz

```
torch_manual_seed(1)
fitted_lin = mnist_linear |>
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_sgd,
    metrics = list(luz_metric_accuracy())
  ) |>
  set_opt_hparams(lr = 0.05) |>
  fit(
    list(X_train_t, y_train_t),
    valid_data = list(X_test_t, y_test_t),
    epochs = 30,
    dataloader_options = list(batch_size = 64, shuffle = TRUE),
    verbose = FALSE
  )
```

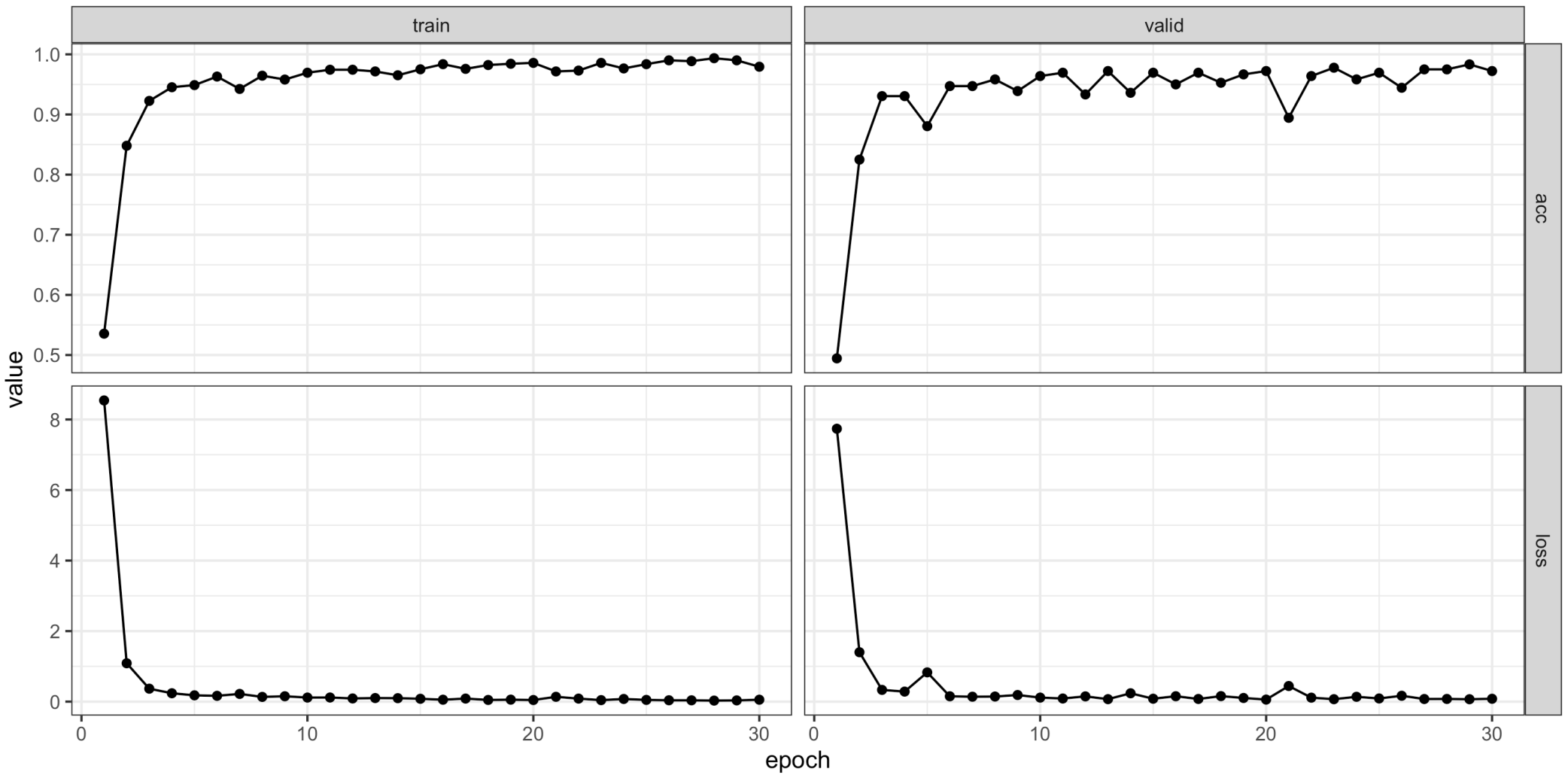
```
get_metrics(fitted_lin) |> filter(set == "t
```

```
    set metric epoch      value
57 train   loss    29 0.03515765
58 train    acc    29 0.99005682
59 train   loss    30 0.05474032
60 train    acc    30 0.97940341
```

```
get_metrics(fitted_lin) |> filter(set == "va
```

```
    set metric epoch      value
57 valid   loss    29 0.06857039
58 valid    acc    29 0.98333333
59 valid   loss    30 0.08136037
60 valid    acc    30 0.97222222
```

Linear model results



Actication layers

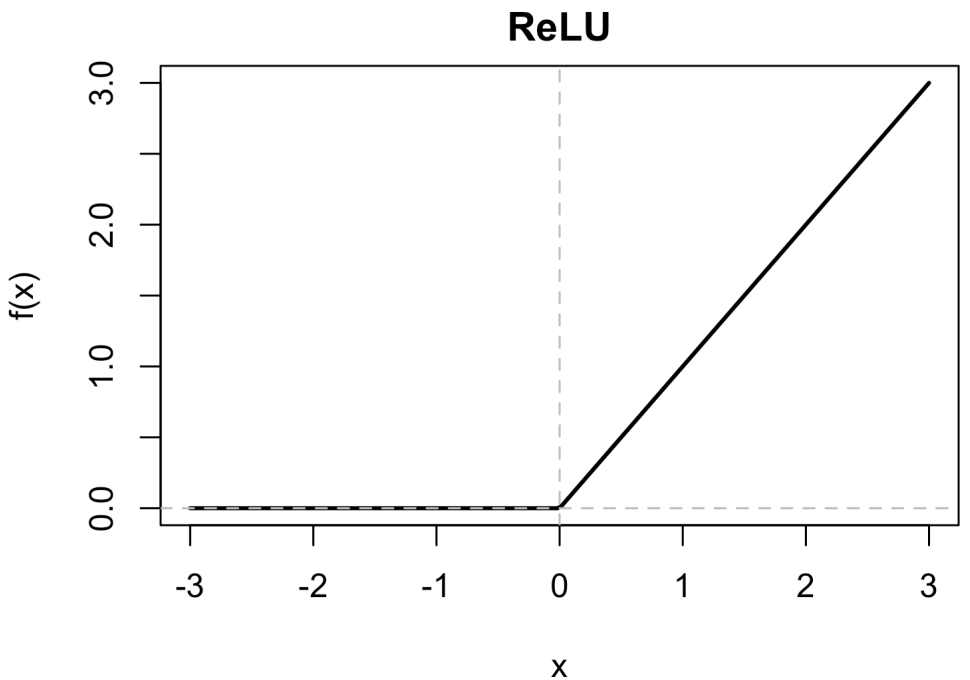
We're about to start building more complex models by stacking layers on top of each other. The issue is that if we just stack `nn_linear` layers, the whole thing collapses back down to a single linear transformation - no matter how many layers we add, the model is still just a linear model.

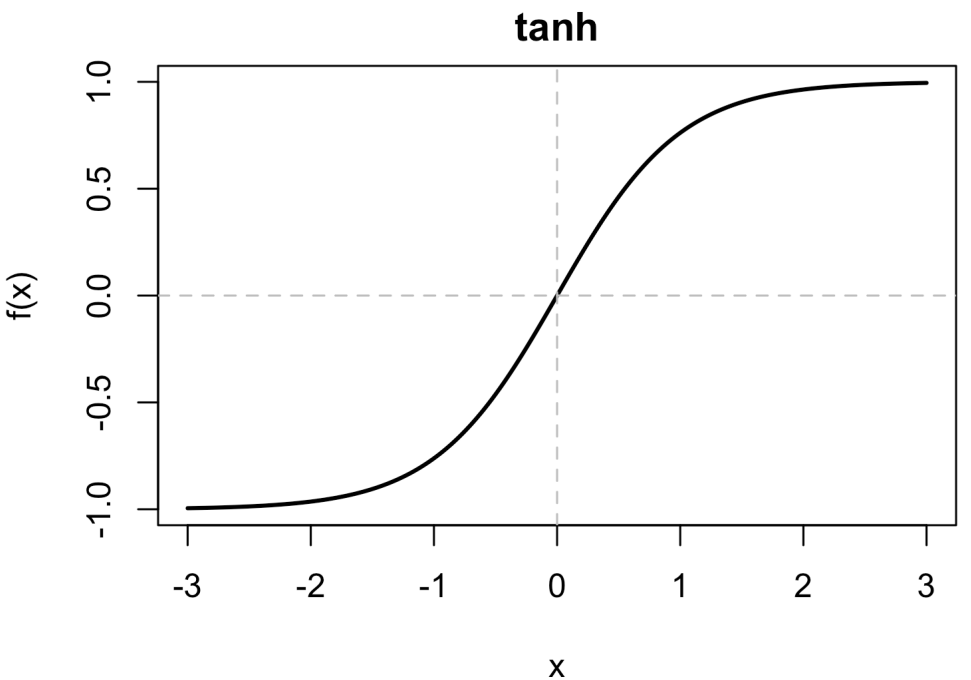
To solve this problem, we need to add a nonlinearity between the linear layers - an *activation function* that is applied element-wise to the output of each linear layer.

The two common choices:

- `nn_relu()` - Rectified Linear Unit: $f(x) = \max(0, x)$. Fast, sparse, and the default in most modern networks.
- `nn_tanh()` - Hyperbolic tangent: $f(x) = \tanh(x)$. Outputs in $(-1, 1)$, symmetric around zero.

Activations in practice





ReLU zeroes out negative inputs (creating sparsity), while tanh squashes everything to $(-1, 1)$. Both are applied

A feed-forward network

Now we'll build a model flexible model by introducing a hidden layer with 64 units and a ReLU activation.

```
mnist_fnn = nn_module(
  initialize = function(hidden = 64, activation = nn_relu) {
    self$net = nn_sequential(
      nn_linear(64, hidden),
      activation(),
      nn_linear(hidden, 10)
    )
  },
  forward = function(x) {
    self$net(x)
  }
)
```

Fitting the FNN

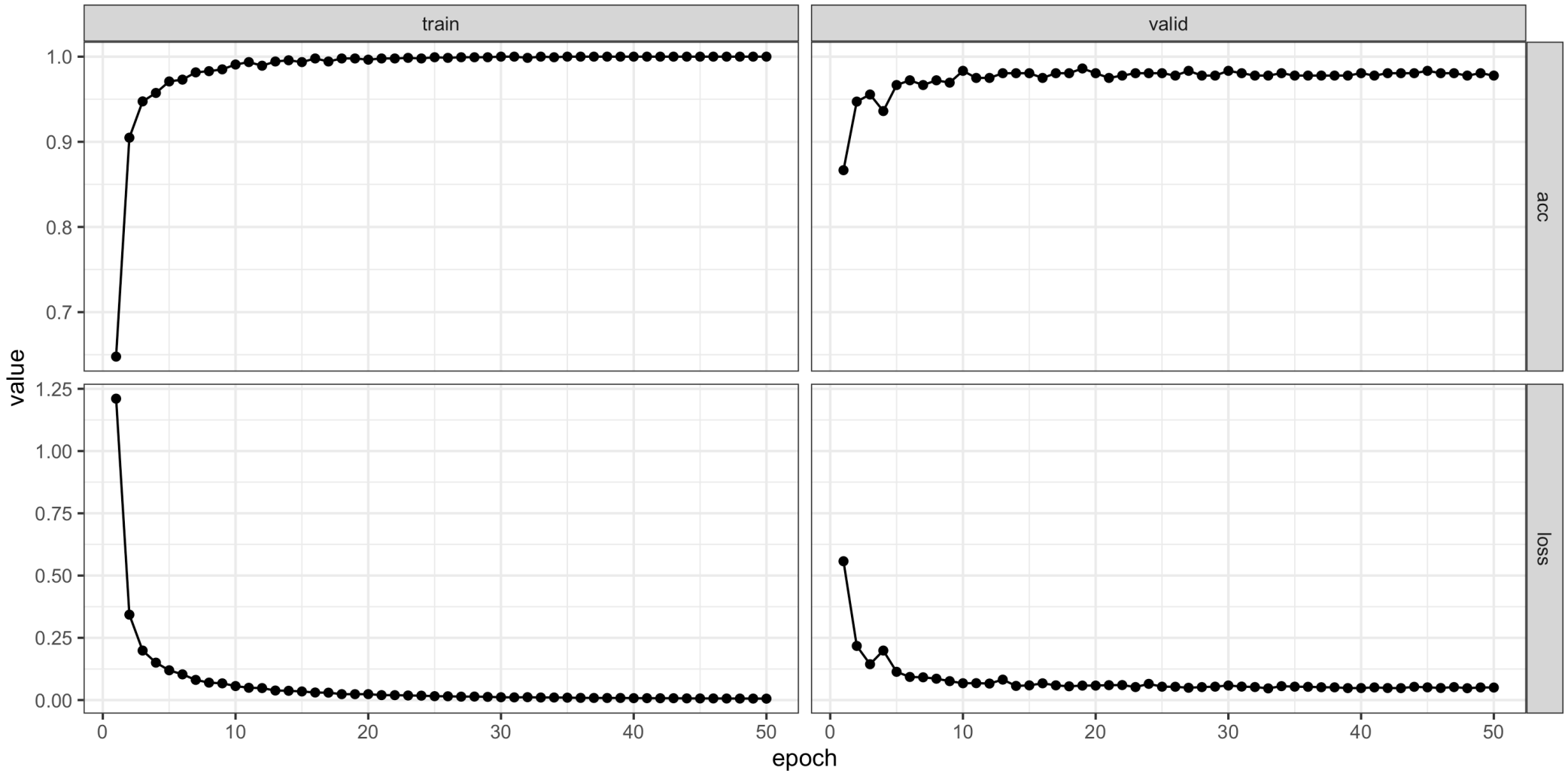
```
torch_manual_seed(1)
fitted_fnn = mnist_fnn |>
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_sgd,
    metrics = list(luz_metric_accuracy())
  ) |>
  set_hparams(hidden = 64, activation = nn_relu) |>
  set_opt_hparams(lr = 0.05) |>
  fit(
    list(X_train_t, y_train_t),
    valid_data = list(X_test_t, y_test_t),
    epochs = 50,
    dataloader_options = list(batch_size = 64, shuffle = TRUE),
    verbose = FALSE
  )
```

```
get_metrics(fitted_fnn) |>
  filter(set == "train") |> tail(4)
```

```
     set metric epoch       value
97  train   loss    49 0.005963060
98  train    acc    49 1.000000000
99  train   loss    50 0.005669653
100 train    acc    50 1.000000000
```

```
get_metrics(fitted_fnn) |>
  filter(set == "valid") |> tail(4)
```

```
     set metric epoch      value
97  valid   loss    49 0.05062127
98  valid    acc    49 0.98055556
99  valid   loss    50 0.05027586
100 valid    acc    50 0.97777778
```


train
valid
acc
loss
value
epoch
1.0
0.9
0.8
0.7
1.25
1.00
0.75
0.50
0.25
0.00
0
10
20
30
40
50

Swapping the activation

We can switch from ReLU to `nn_tanh` in via `set_hparams()` without rewriting the model:

```
torch_manual_seed(1)
fitted_tanh = mnist_fnn |>
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_sgd,
    metrics = list(luz_metric_accuracy())
  ) |>
  set_hparams(hidden = 64, activation = nn_tanh) |>
  set_opt_hparams(lr = 0.05) |>
  fit(
    list(X_train_t, y_train_t),
    valid_data = list(X_test_t, y_test_t),
    epochs = 50,
    dataloader_options = list(batch_size = 64, shuffle = TRUE),
    verbose = FALSE
  )
```

CIFAR10

The CIFAR10 dataset

- 60,000 color images at 32×32 resolution
- 10 classes - airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck
- 50,000 training / 10,000 test split
- A standard benchmark for evaluating CNN architectures

Smaller brother to the CIFAR100 dataset (100 classes) and the ImageNet dataset (1000 classes, 224×224 resolution) - the latter was the standard benchmark for large-scale image classification.

Loading the cached data

```r
train = readRDS("data/cifar10/train.rds")
test  = readRDS("data/cifar10/test.rds")

str(train, max.level = 1)
```

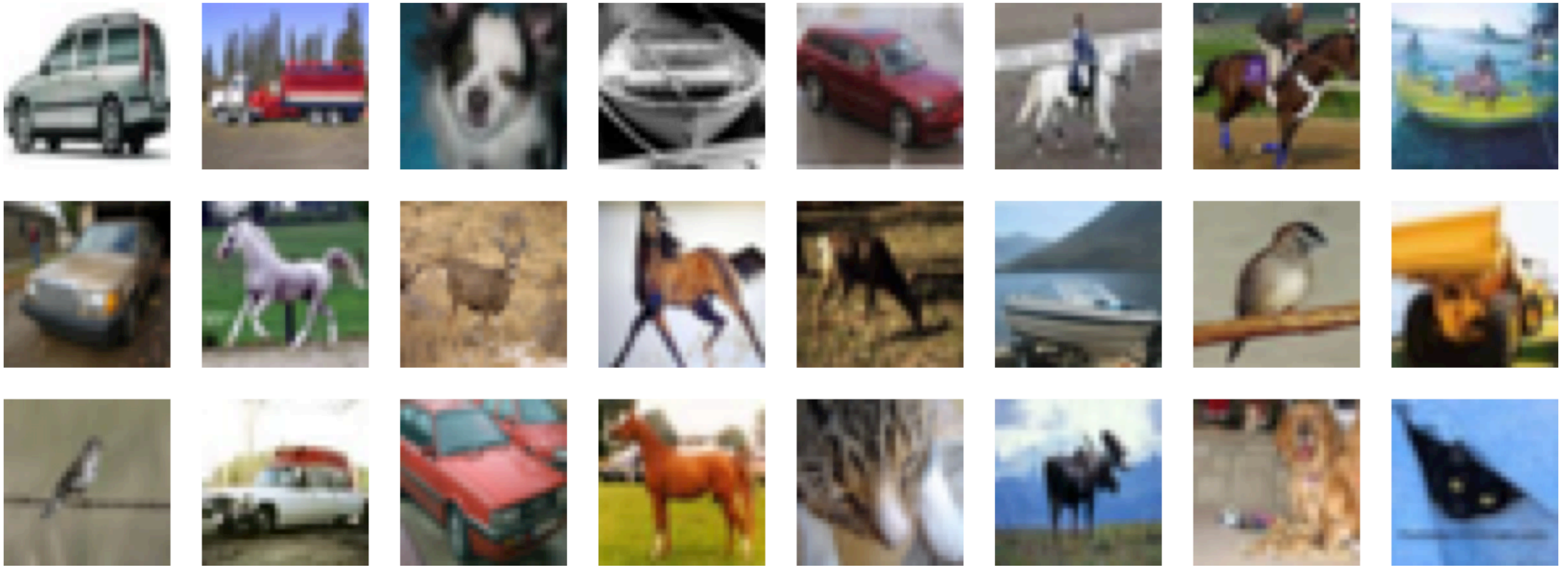
```
List of 3
 $ x      : int [1:50000, 1:3, 1:32, 1:32] 59 154 255 28 170 159 164 28 134 125 ...
  ..- attr(*, "dimnames")=List of 4
 $ y      : int [1:50000] 7 10 10 5 2 2 3 8 9 4 ...
 $ classes: chr [1:10] "airplane" "automobile" "bird" "cat" ...
```

```r
train$classes
```

```
 [1] "airplane"   "automobile" "bird"       "cat"        "deer"
 [6] "dog"        "frog"       "horse"      "ship"       "truck"
```

`x` is stored as an `integer` array of shape `(n, 3, 32, 32)` with pixel values in `0:255`. We will rescale to `[0, 1]` floats when

Example images



A custom `dataset()`

For datasets that don't fit neatly into an `(x, y)` tuple we define a `dataset()` class with `initialize`, `.getitem`, and `.length` methods - the R torch analogue of Python's `torch.utils.data.Dataset`:

```
cifar10_dataset = dataset(
  name = "cifar10",
  initialize = function(data) {
    self$x = torch_tensor(data$x, dtype = torch_float()) / 255
    self$y = torch_tensor(data$y, dtype = torch_long())
  },
  .getitem = function(i) {
    list(x = self$x[i, , , ], y = self$y[i])
  },
  .length = function() {
    self$y$size(1)
  }
)
```

From dataset to dataloader

```
train_ds = cifar10_dataset(train)
test_ds  = cifar10_dataset(test)

train_dl = dataloader(train_ds, batch_size = 128, shuffle = TRUE)
test_dl  = dataloader(test_ds,  batch_size = 128, shuffle = FALSE)

length(train_dl)
```

```
[1] 391
```

```
batch = train_dl$.iter()$.next()
batch$x$shape
```

```
[1] 128   3  32  32
```

```
batch$y$shape
```

```
[1] 128
```

A CIFAR10 feed-forward model

As a baseline, we flatten each 32×32×3 image into a 3072-length vector and feed it through a two-hidden-layer network - the same approach we used for MNIST digits.

```
cifar_fnn = nn_module(
  initialize = function() {
    self$net = nn_sequential(
      nn_flatten(),
      nn_linear(3 * 32 * 32, 512),
      nn_relu(),
      nn_linear(512, 128),
      nn_relu(),
      nn_linear(128, 10)
    )
  },
  forward = function(x) {
    self$net(x)
  }
)
```

Fitting the CIFAR10 FNN

When the inputs are dataloaders, luz's `fit()` iterates over them directly - no `dataloader_options` needed.

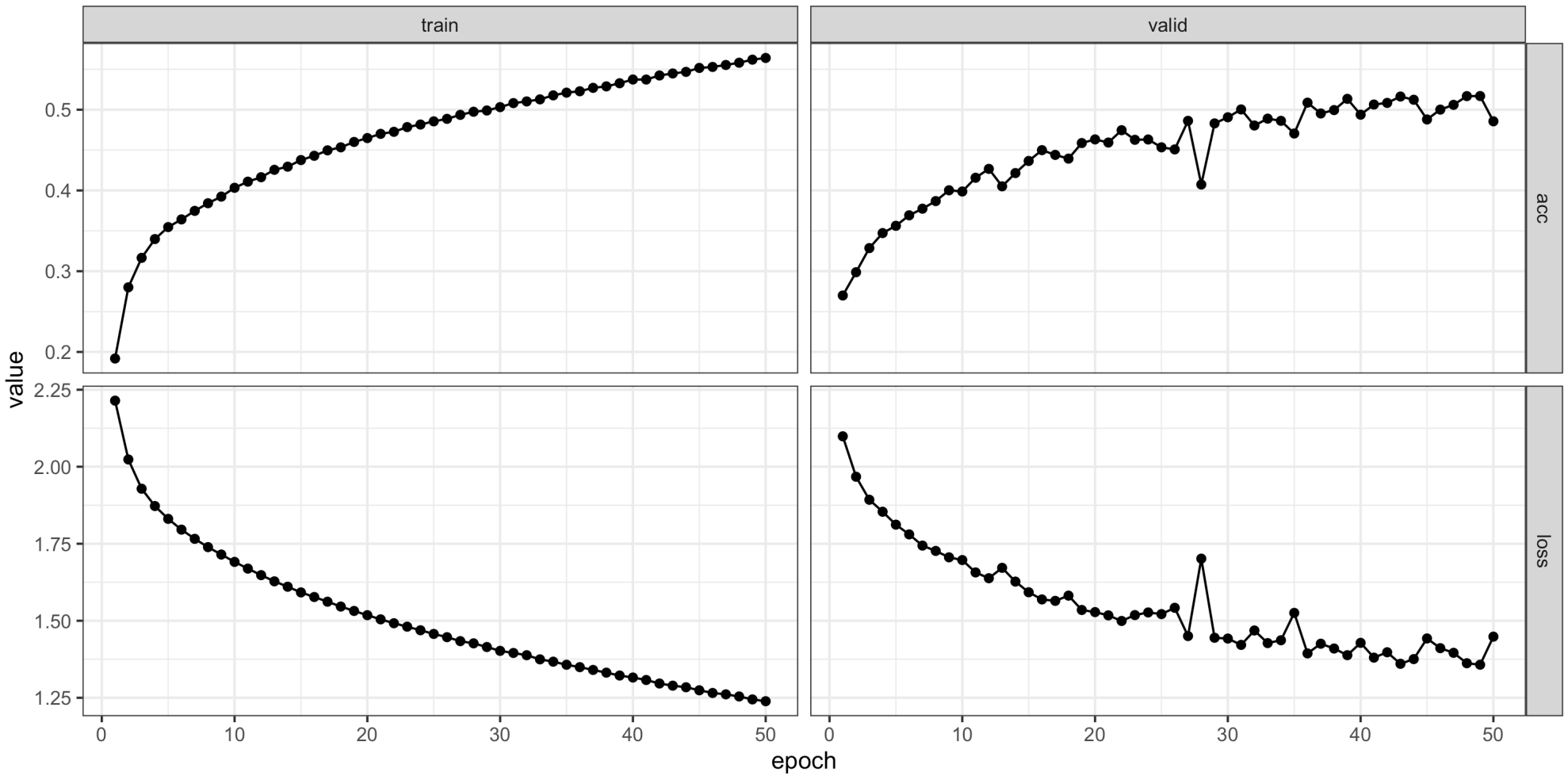
```
torch_manual_seed(1)
fitted_cifar_fnn = cifar_fnn |>
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_sgd,
    metrics = list(luz_metric_accuracy())
  ) |>
  set_opt_hparams(lr = 0.01) |>
  fit(
    train_dl,
    valid_data = test_dl,
    epochs = 50,
    verbose = FALSE
  )
```

```
get_metrics(fitted_cifar_fnn) |>
  filter(set == "train") |> tail(4)
```

```
      set metric epoch    value
97  train   loss    49 1.244670
98  train    acc    49 0.561940
99  train   loss    50 1.238834
100 train    acc    50 0.564060
```

```
get_metrics(fitted_cifar_fnn) |>
  filter(set == "valid") |> tail(4)
```

```
      set metric epoch    value
97  valid   loss    49 1.357298
98  valid    acc    49 0.516900
99  valid   loss    50 1.448470
100 valid    acc    50 0.485600
```

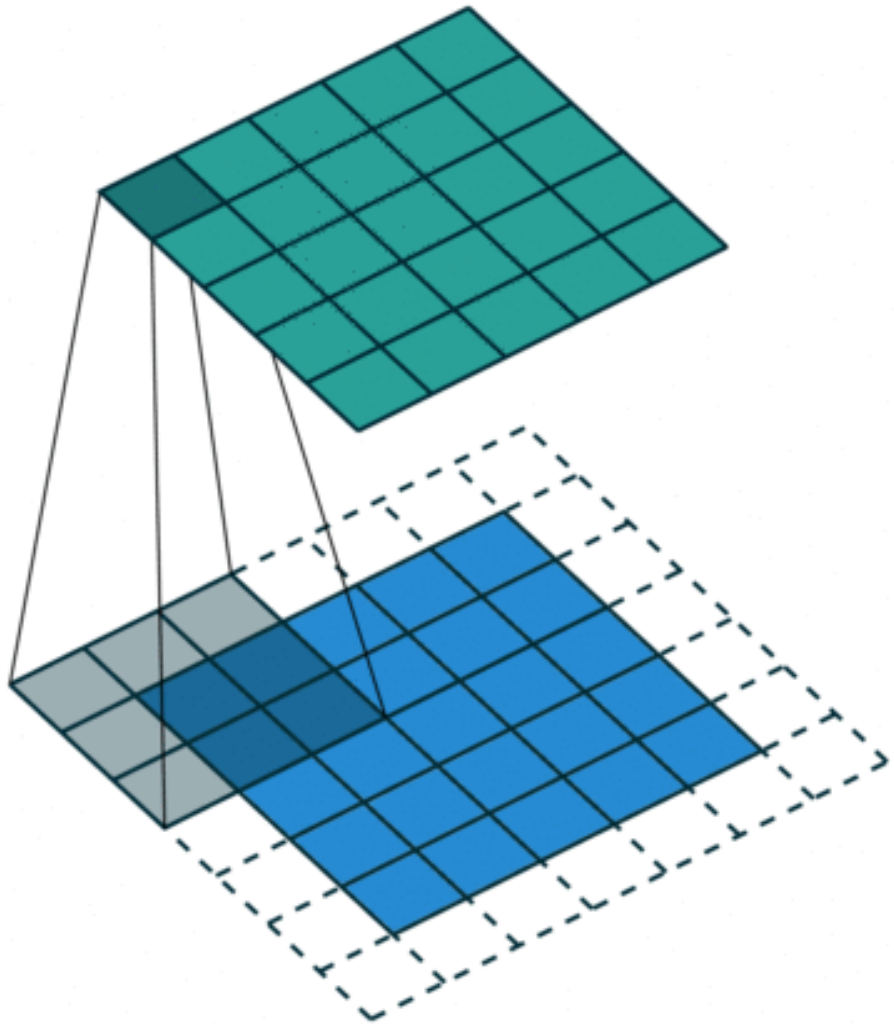

train
valid
acc
loss
value
epoch
0.2
0.3
0.4
0.5
1.25
1.50
1.75
2.00
2.25
0
10
20
30
40
50

2D convolutions

The model we just used flattened all of the data which throws away all spatial structure - the model has no way to know that nearby pixels are related. This limits how well a fully connected network can do on image data.

3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0



nn_conv2d

A 2D convolutional layer slides a bank of learnable filters over the spatial dimensions of its input - the same filter weights are reused at every location:

```
nn_conv2d(in_channels, out_channels, kernel_size, padding = 0, stride = 1)
```

- Input: `(N, in_channels, H, W)` - batch of multi-channel images
- Output: `(N, out_channels, H', W')` - one feature map per output channel
- Parameters: $\texttt{out\_channels} \times (\texttt{in\_channels} \times \texttt{kernel\_size}^2 + 1)$

nn_conv2d in practice

```
conv = nn_conv2d(in_channels = 3, out_channels = 8,
                 kernel_size = 3, padding = 1)
conv$weight$shape
```

```
[1] 8 3 3 3
```

```
conv$bias$shape
```

```
[1] 8
```

```
x = torch_randn(5, 3, 32, 32)
conv(x)$shape
```

```
[1]  5  8 32 32
```

Pooling

After a convolutional layer we typically *downsample* the features. Pooling layers summarize small spatial neighborhoods into a single value - this has no learnable parameters.

Why pool?

- Reduces spatial dimensions, cutting computation and memory for later layers
- Gives a degree of translation invariance - small shifts in the input produce the same pooled output
- Forces the network to learn coarser, more abstract features as we go deeper

`nn_max_pool2d` and `nn_avg_pool2d`

The two common pooling operations over a 2×2 window:

- `nn_max_pool2d(kernel_size = 2)` - takes the maximum value in each window
- `nn_avg_pool2d(kernel_size = 2)` - takes the mean of values in each window

Both halve the spatial dimensions while leaving the channel count unchanged.

```
x = torch_randn(5, 8, 32, 32)

nn_max_pool2d(kernel_size = 2)(x)$shape
```

```
[1]  5  8 16 16
```

```
nn_avg_pool2d(kernel_size = 2)(x)$shape
```

```
[1]  5  8 16 16
```

A CIFAR10 CNN

Directly adapted from the PyTorch CIFAR10 tutorial - two conv+pool blocks feeding into a 3-layer mlp with ReLU activations.

```
cifar_cnn = nn_module(
  initialize = function() {
    self$net = nn_sequential(       # (N, 3, 32, 32)
      nn_conv2d(3, 6, kernel_size = 5),  # -> (N, 6, 28, 28)
      nn_relu(),
      nn_max_pool2d(kernel_size = 2, stride = 2),  # -> (N, 6, 14, 14)
      nn_conv2d(6, 16, kernel_size = 5), # -> (N, 16, 10, 10)
      nn_relu(),
      nn_max_pool2d(kernel_size = 2, stride = 2),  # -> (N, 16, 5, 5)
      nn_flatten(),                       # -> (N, 400)
      nn_linear(16 * 5 * 5, 120),       # -> (N, 120)
      nn_relu(),
      nn_linear(120, 84),               # -> (N, 84)
      nn_relu(),
      nn_linear(84, 10)                 # -> (N, 10)
    )
  },
  forward = function(x) {
    self$net(x)
  }
)
```

Training the CNN

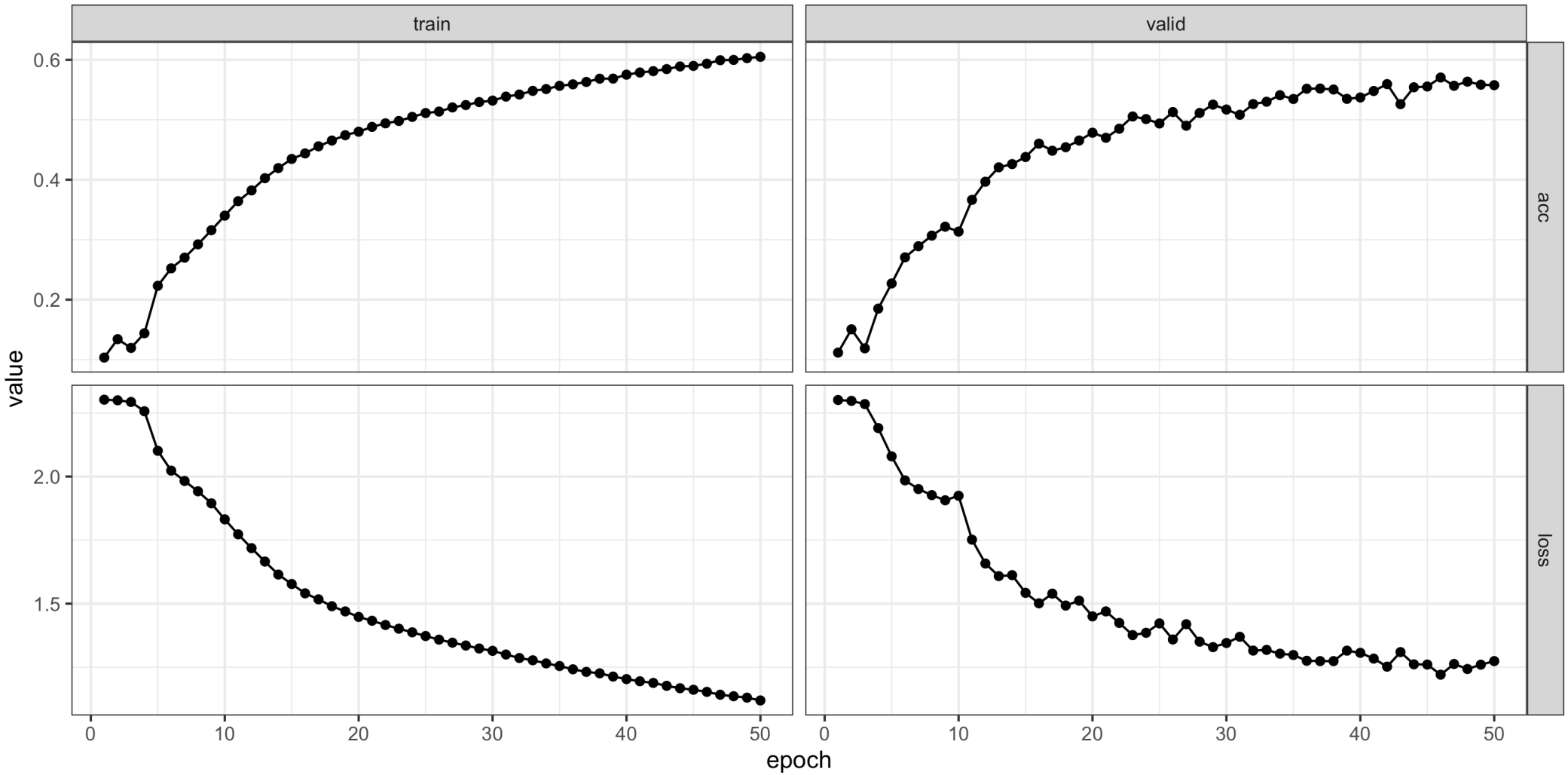
```
torch_manual_seed(1)
fitted_cifar = cifar_cnn |>
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_sgd,
    metrics = list(luz_metric_accuracy())
  ) |>
  set_opt_hparams(lr = 0.01) |>
  fit(
    train_dl,
    valid_data = test_dl,
    epochs = 50,
    verbose = FALSE
  )
```

```
get_metrics(fitted_cifar) |>
  filter(set == "train") |> tail(4)
```

```
      set metric epoch    value
97  train   loss    49 1.130158
98  train    acc    49 0.602800
99  train   loss    50 1.119145
100 train    acc    50 0.605060
```

```
get_metrics(fitted_cifar) |>
  filter(set == "valid") |> tail(4)
```

```
      set metric epoch    value
97  valid   loss    49 1.259922
98  valid    acc    49 0.558500
99  valid   loss    50 1.273951
100 valid    acc    50 0.557700
```


train
valid
acc
loss
value
epoch
0.6
0.4
0.2
2.0
1.5
0
10
20
30
40
50

Going further - VGG16

The VGG16 model

VGG16 was state of the art on ImageNet in 2014. It is a simple repeating pattern of `Conv → BatchNorm → ReLU` blocks interleaved with max pooling, finished with a single linear layer. Implementing it is short - `make_layers()` walks a config list and builds the sequential model for us.

```
vgg16 = nn_module(
  initialize = function() {
    cfg = c(64, 64, "M", 128, 128, "M", 256, 256, 256, "M",
            512, 512, 512, "M", 512, 512, 512, "M")
    in_channels = 3
    layers = list()
    for (x in cfg) {
      if (x == "M") {
        layers[[length(layers) + 1]] = nn_max_pool2d(kernel_size = 2, stride = 2)
      } else {
        x = as.integer(x)
        layers[[length(layers) + 1]] = nn_conv2d(in_channels, x,
                                                 kernel_size = 3, padding = 1)
        layers[[length(layers) + 1]] = nn_batch_norm2d(x)
        layers[[length(layers) + 1]] = nn_relu(inplace = TRUE)
        in_channels = x
      }
    }
    layers[[length(layers) + 1]] = nn_flatten()
    layers[[length(layers) + 1]] = nn_linear(512, 10)
    self$net = do.call(nn_sequential, layers)
  },
  forward = function(x) {
    self$net(x)
  }
)
```

Fitting VGG16

On a laptops, each epoch of VGG16 on the full CIFAR10 training set takes several minutes on CPU. The code below would produce roughly the following loss / accuracy with `lr = 0.01`:

Code | Output

```
torch_manual_seed(1)
fitted_vgg = vgg16 |>
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_sgd,
    metrics = list(luz_metric_accuracy())
  ) |>
  set_opt_hparams(lr = 0.01) |>
  fit(
    train_dl,
    valid_data = test_dl,
    epochs = 10
  )
```

Where to go next

- `torch` - the full low-level API. Everything we have done translates one-for-one with PyTorch documentation.
- `luz` - higher-level training, callbacks, early stopping, learning-rate schedulers, checkpointing.
- `torchvision` - image datasets, transforms, and a growing set of pretrained models (ResNet, VGG, EfficientNet, …).
- `torchdatasets` - a growing set of tabular and vision datasets with ready-made `dataset()` classes.
- GPU support - everything here runs unchanged on a CUDA-capable machine; tensors and modules are moved with `$to(device = "cuda")`.
  - Departmental Servers have 2x NVidia A4000 GPUs each and you are welcome to make use of them for learning purposes