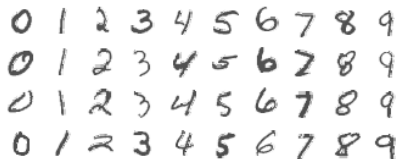


10.2 Multilayer Neural Networks

Multilayer Neural Networks

- ▶ In theory, a single hidden layer with a large number of units has the ability to approximate most functions.
- ▶ However, the learning task of discovering a good solution is easier with multiple layers of modest size.
- ▶ In practice, neural networks typically have more than one hidden layer.

Example: MNIST Digits



Handwritten digits

28×28 grayscale images

60K train, 10K test images

Features are the 784 pixel
grayscale values $\in (0, 255)$

Labels are the digit class 0–9

- Goal: build a classifier to predict the digit.

Example: MNIST Digits

- ▶ Two layer-network with 256 units at first layer, 128 at second layer.
- ▶ Ten output variables, each representing a digit class.
 - ▶ In *multi-task learning*, we can predict different responses simultaneously with a single network; each has input in the formation of the hidden layers.
 - ▶ The loss function is tailored for the multiclass classification task.

Example: MNIST Digits

The first hidden layer is as before, with

$$\begin{aligned} A_k^{(1)} &= h_k^{(1)}(X) \\ &= g \left(w_{k0}^{(1)} + \sum_{j=1}^p w_{kj}^{(1)} x_j \right) \end{aligned}$$

for $k = 1, \dots, K_1$

Example: MNIST Digits

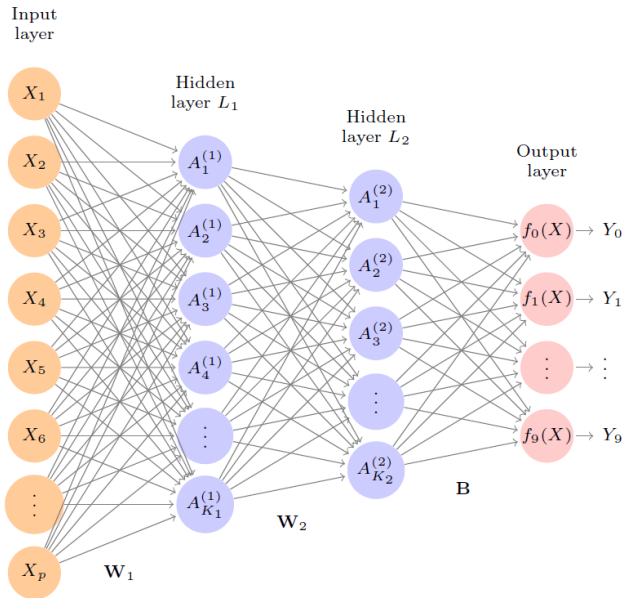
The second hidden layer takes the activations $A_k^{(1)}$ as inputs and computes new activations

$$\begin{aligned} A_l^{(2)} &= h_l^{(2)}(X) \\ &= g \left(w_{l0}^{(2)} + \sum_{k=1}^{K_1} w_{lj}^{(2)} A_k^{(1)} \right) \end{aligned}$$

for $l = 1, \dots, K_2$

Example: MNIST Digits

- ▶ The first hidden layer is made up directly of functions of X .
- ▶ Each subsequent hidden layer is a function of the prior layer's activations.
- ▶ So a chain of hidden layers builds up complex transformations of X that ultimately feed into the output layer as features.



Example: MNIST Digits

- ▶ The input has $p = 784$ units (each pixel is a feature);
- ▶ $K_1 = 256$ units; $K_2 = 128$ units.
- ▶ Matrix W_1 contains the coefficients $w^{(1)}$ and has dimensions $785 \times 256 = 200,960$
- ▶ Matrix W_2 contains the coefficients $w^{(2)}$ and has dimensions $257 \times 128 = 32,896$

Example: MNIST Digits

The final hidden layer feeds into the output layer, which now has ten responses.

This requires computing ten different linear models

$$\begin{aligned} Z_m &= \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} h_l^{(2)}(X) \\ &= \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} A_l^{(2)} \end{aligned}$$

for $(m=0, 1, \dots, 9)$

- ▶ The matrix B stores all $129 \times 10 = 1290$ of these weights (parameters).
- ▶ There are 235,146 total parameters in this model!

Qualitative Responses

- ▶ If these were separate quantitative responses, we could just set $f_m(X) = Z_m$
- ▶ However, we would prefer our estimates to represent class probabilities $f_m(X) = P(Y = m|X)$ (as with multinomial logistic regression)
- ▶ To do this, we use the *softmax* activation function

$$f_m(X) = P(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^9 e^{Z_l}}$$

- ▶ The classifier then assigns the image to the class with the highest probability.

Training the Network

We find coefficients that minimize the negative multinomial log-likelihood

$$-\sum_{i=1}^n \sum_{m=0}^9 y_{im} \log[f_m(x_i)]$$

- ▶ This is also known as *cross-entropy*.
- ▶ We will see more details in a later section.

Example: MNIST Digist

The next few slides cover code found in the Chapter 10 lab.

```
library(torch)
library(luz) # high-level interface for torch
library(torchvision) # for datasets and image transformation
library(torchdatasets) # for datasets we are going to use
library(zeallot)
torch_manual_seed(0)
```

Load Data from Package

```
train_ds <- mnist_dataset(root = ".", train = TRUE, download = TRUE)
test_ds  <- mnist_dataset(root = ".", train = FALSE, download = TRUE)
```

Load Data from Package

```
str(train_ds[1])
```

```
## List of 2  
## $ x: int [1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 0 0 ...  
## $ y: int 6
```

```
str(test_ds[2])
```

```
## List of 2  
## $ x: int [1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 0 0 ...  
## $ y: int 3
```

```
length(train_ds)
```

```
## [1] 60000
```

```
length(test_ds)
```

```
## [1] 10000
```

```
transform <- function(x) {  
  x %>%  
    transform_to_tensor() # returns a 1 x 28 x 28 tensor scaled to [0, 1]  
}  
train_ds <- mnist_dataset(  
  root = ".",  
  train = TRUE,  
  download = TRUE,  
  transform = transform # use transform function def above  
)  
test_ds <- mnist_dataset(  
  root = ".",  
  train = FALSE,  
  download = TRUE,  
  transform = transform  
)
```

```
modelnn <- nn_module(  
  initialize = function() {  
    self$linear1 <- nn_linear(in_features = 28*28, out_features = 256)  
    self$linear2 <- nn_linear(in_features = 256, out_features = 128)  
    self$linear3 <- nn_linear(in_features = 128, out_features = 10)  
    self$drop1 <- nn_dropout(p = 0.4)  
    self$drop2 <- nn_dropout(p = 0.3)  
    self$activation <- nn_relu()  
  },  
  forward = function(x) {  
    x %>%  
      torch_flatten(start_dim = 2) %>%  
      self$linear1() %>% self$activation() %>% self$drop1() %>%  
      self$linear2() %>% self$activation() %>% self$drop2() %>%  
      self$linear3()  
  }  
)
```

Details

- ▶ `initialize` used to specify all layers used in the model.
 - ▶ Takes us from input -> hidden 1 -> hidden 2 -> output
 - ▶ `nn_linear(784, 256)` defines a dense layer that goes from
 - ▶ `nn_dropout()` defines dropout layers
 - ▶ used to perform dropout regularization, which we will talk about in a later section
 - ▶ Activation function defined by `nn_relu()`
- ▶ `forward()` used to set the order in which these layers are called
 - ▶ we call them in blocks: linear, activation, dropout
 - ▶ last layer that does not use an activation function or dropout

Confirm Model Set Up

```
print(modelnn())
```

```
## An `nn_module` containing 235,146 parameters.
```

```
##
```

```
## -- Modules -----
```

```
## * linear1: <nn_linear> #200,960 parameters
```

```
## * linear2: <nn_linear> #32,896 parameters
```

```
## * linear3: <nn_linear> #1,290 parameters
```

```
## * drop1: <nn_dropout> #0 parameters
```

```
## * drop2: <nn_dropout> #0 parameters
```

```
## * activation: <nn_relu> #0 parameters
```

Add Details

```
modelnn <- modelnn %>%  
  setup(  
    loss = nn_cross_entropy_loss(), ## specify cross-entropy loss  
    optimizer = optim_rmsprop,  
    metrics = list(luz_metric_accuracy())  
  )
```

Fit The Model

```
system.time(  
  fitted <- modelnn %>%  
    fit(  
      data = train_ds,  
      epochs = 10, #15,  
      valid_data = 0.2,  
      dataloader_options = list(batch_size = 256),  
      verbose = TRUE  
    )  
)
```

Epoch 1/10

Train metrics: Loss: 1.393 - Acc: 0.7492

Valid metrics: Loss: 0.3149 - Acc: 0.912

Epoch 2/10

Train metrics: Loss: 0.441 - Acc: 0.8662

Valid metrics: Loss: 0.2534 - Acc: 0.9273

plot(fitted)

