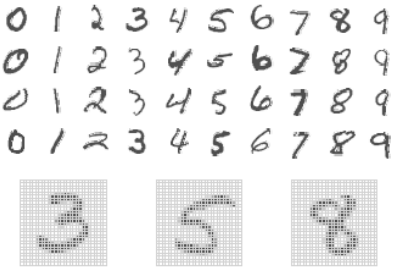
10.2 Multilayer Neural Networks

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 $k = 1, \ldots, K_1$

Example: MNIST Digits

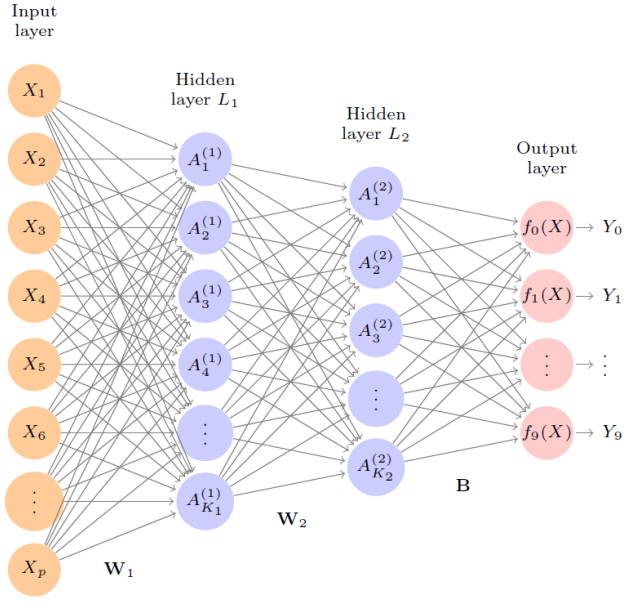
The second hidden layer takes the activations $A_k^{(1)}$ as inputs and computes new activations

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

for $l = 1, \ldots, K_2$

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.


Input layer
Hidden layer L_1
Hidden layer L_2
Output layer
X_1
X_2
X_3
X_4
X_5
X_6
X_p
$A_1^{(1)}$
$A_2^{(1)}$
$A_3^{(1)}$
$A_4^{(1)}$
$A_{K_1}^{(1)}$
$A_1^{(2)}$
$A_2^{(2)}$
$A_3^{(2)}$
$A_{K_2}^{(2)}$
$f_0(X) \to Y_0$
$f_1(X) \to Y_1$
$f_9(X) \to Y_9$
$\mathbf{W}_1$
$\mathbf{W}_2$
$\mathbf{B}$

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

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

for (m=0, 1, ..., 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 he *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 ...
##  $ 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 ...
##  $ y: int 3
```

```
length(train_ds)
```

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

```
length(test_ds)
```

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

```
transform <- function(x) {
  x %>%
    torch_tensor() %>%  # convert image to tensor
    torch_div(255) # rescale to unit interval <greyscale values in (0,255)
}
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$fc1 <- nn_linear(28 * 28, 256)
    self$fc2 <- nn_linear(256, 128)
    self$fc3 <- nn_linear(128, 10)
    self$drop1 <- nn_dropout(p = 0.4)
    self$drop2 <- nn_dropout(p = 0.3)
  },
  forward = function(x) {
    # if batch comes in as 28 x 28 x batch_size, move batch to front
    if (length(x$shape) == 3){ x <- x$permute(c(3, 1, 2)) }
    x <- x$flatten(start_dim = 2)
    x %>%
      self$fc1() %>% nnf_relu() %>% self$drop1() %>%
      self$fc2() %>% nnf_relu() %>% self$drop2() %>%
      self$fc3()
  }
)
```

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 -----------------------------------------------------------------
## * fc1: <nn_linear> #200,960 parameters
## * fc2: <nn_linear> #32,896 parameters
## * fc3: <nn_linear> #1,290 parameters
## * drop1: <nn_dropout> #0 parameters
## * drop2: <nn_dropout> #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: 1.2409 - Acc: 0.7823
## Valid metrics: Loss: 0.2848 - Acc: 0.9241
## Epoch 2/10
## Train metrics: Loss: 0.3718 - Acc: 0.8926
## Valid metrics: Loss: 0.2356 - Acc: 0.9305
```

```
plot(fitted)
```