Simple Convolutional Neural Network From Scratch

Chris Yogodzinski

Notebook Overview

This notebook is a follow up to src/python/scratch_two_layer.ipynb.
The previous notebook I made a simple neural network from scratch to predict handwritten digits in the mnist dataset. The neural network consisted of one fully connected hidden layer and an output layer.

In this notebook I will be adding a convolutional layer and pooling layer to this simple neural network. The convolutions in this neural network function to key features from the dataset prior to the predictive fully connected layer.

Variables

I matrix: a mnist image of dim $= (m \text{ by } n)$
K matrix: kernel of dim $= (m_k \text{ by } n_k)$
s integer: stride, a constant set by user
ϕ_a function: activation function of convolution
F matrix: a feature map of dim $= (m - m_k + 1 \text{ by } n - n_k + 1)$ or $(i \text{ by } j)$
ϕ_p function: pooling function applied to feature map
P matrix: pooled feature map, condensed image of dim $= (\frac{m-m_k}{s} \text{ by } \frac{n-n_k}{s})$

Convolutional Layer

The convolutional layer in a cnn is named after a method known as kernel convolution.
Kernel convolution works by passing a filter or kernal iteratively over the input matrix and transform the values of the input based on the values of the kernel.
In this notebook our kernal transformation will be $\sum_i \sum_j I[m-i, n-j] \times K$.
In words this is element-wise multiplication of the kernel-sized subset of I and K followed by the sum of the product.
Kernel Convolution simplifies our image which reduces the number of parameters our model needs to learn in the fully connected layers.
However the kernel itself is also a set of paramters that needs to be learned so that we're not losing important information when we simplify the images.

Importing the data

The following code chunks read in the binary formatted mnist images

```r
setwd("~/Projects/nn_playground")

read_mnist <- function(label_path, im_path) {
    # read in labels
    f <- file(label_path, "rb")
    meta <- readBin(f, n = 2, "integer", endian = "big")
    labels <- as.integer(readBin(f, n = meta[2], "raw", endian = "big"))
    close(f)

    # read in imgs
    f <- file(im_path, "rb")
    meta <- readBin(f, n = 4, "integer", endian = "big")
    byte_count <- meta[2] * meta[3] * meta[4]
    imgs <- readBin(f, n = byte_count, "raw", endian = "big")
    close(f)
    imgs <- array(as.integer(imgs), dim = c(meta[3], meta[4], meta[2]))
    dat <- list(labels = labels, imgs = imgs)
    return(dat)
}

dat_path = "data/mnist"
train_dat <- read_mnist(
  paste(dat_path, "train-labels.idx1-ubyte", sep = "/"),
  paste(dat_path, "train-images.idx3-ubyte", sep = "/")
)

test_dat <- read_mnist(
  paste(dat_path, "t10k-labels.idx1-ubyte", sep = "/"),
  paste(dat_path, "t10k-images.idx3-ubyte", sep = "/")
)
```

```r
relu <- function(vec) {
  ifelse(vec > 0, vec, 0)
}

relu_deriv <- function(vec) {
  ifelse(vec > 0, 1, 0)
}

softmax <- function(vec) {
  expo <- exp(vec - max(vec))
  expo / sum(expo)
}
```

Image Preprocessing

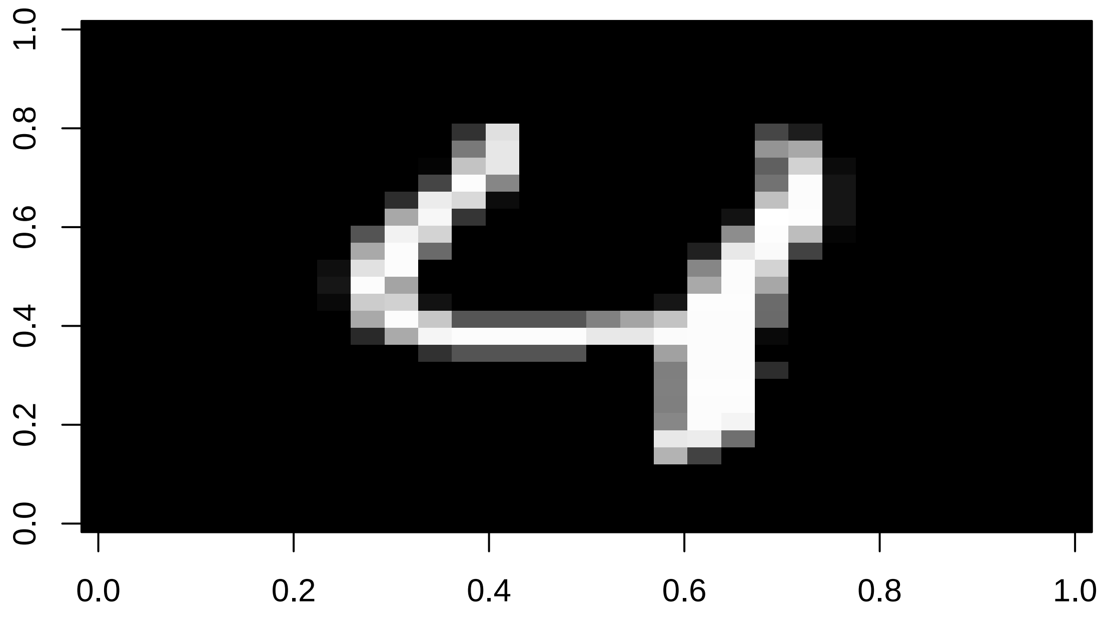
Flipping The Images

The function above reads the images in upside down for some reason.
In the code chunk below I flip the images right side up and visualize them. ### Adding padding to images
In this step I'm adding a border of empty pixels around the edge of the images.
The purpose of this step is to allow our model to get a better "view" of the features around the edges of the image. It is likely this is unnecessary for such simple images like the mnist dataset. In this step I am also scaling the pixel values to be between 0 and 1.

```r
preprocess_img <- function(img) {

  flip_img <- function(img) {
    flipped <- img[, ncol(img):1]
    return(flipped)
  }
  zero_pad_img <- function(img) {
    top_bottom <- rep_len(0, ncol(img))
    img <- rbind(top_bottom, img, top_bottom)
    left_right <- rep_len(0, nrow(img))
    img <- unname(cbind(left_right, img, left_right))
    return(img)
  }
  preproccessed_img <- zero_pad_img(flip_img(img)) / 255

  return(preproccessed_img)
}

preprocess_all <- function(img_array) {
  dims <- dim(img_array) + c(2, 2, 0)
  preprocessed <- array(apply(img_array, 3, preprocess_img), dim = dims)
  return(preprocessed)
}

# apply preprocessing
train_dat[["imgs"]] <- preprocess_all(train_dat[["imgs"]])
test_dat[["imgs"]] <- preprocess_all(test_dat[["imgs"]])

im_num <- 5
# test image
img <- test_dat[["imgs"]][ , , im_num]
cat(test_dat[["labels"]][im_num], sep = "\n")
```

```
## 4
```

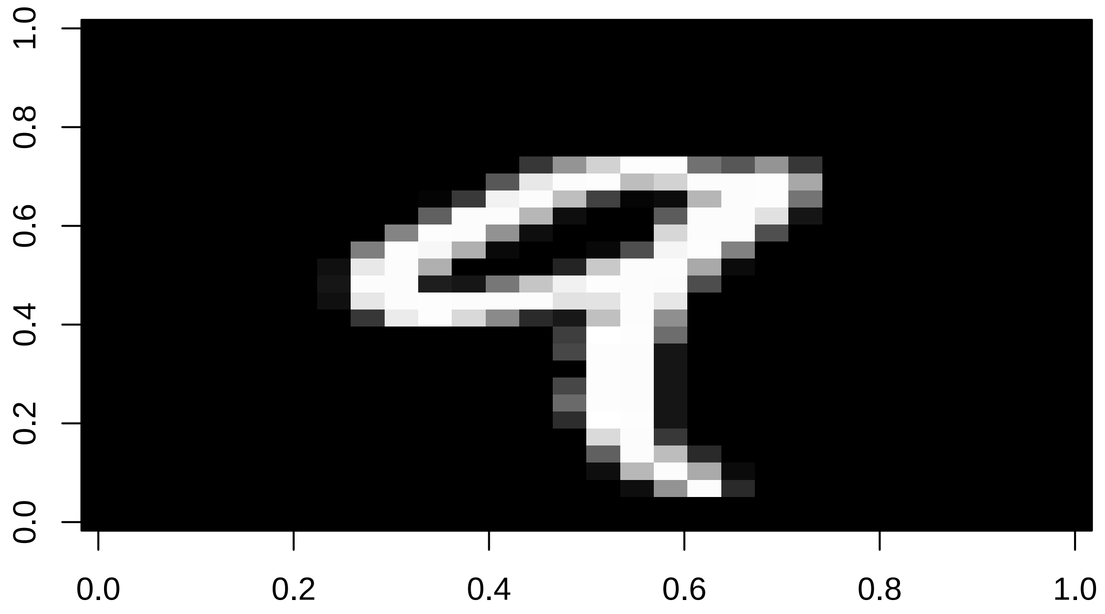
```r
image(img, col = gray((0:255) / 255))
```



```r
# train image
img <- train_dat[["imgs"]][ , , im_num]
cat(train_dat[["labels"]][im_num], sep = "\n")
```

```
## 9
```

```r
image(img, col = gray((0:255) / 255))
```



Forward Propagation

In the following cells i've written some functions needed for forward propagtion

Kernel Convlution

In the cell below I've written some functions to apply kernel convolution to an array of our images. You can see from the images printed at below the cell how some features of the digits have been filtered out after kernel convolution.
Initially I implemented kernel convolution in the spatial domain. Using this procedure kernel convolution of the complete training dataset would take roughly 40 mins.
I could implement some parallel processing or train on a gpu to increase performance.
Instead I decided to implement kernel convolution in the spatial domain using fast fourier transform. This drastically improves performance.

```r
kernel_conv_slow <- function(img, kernel, kernel_dim, feat_map_idx, dim_out, activation_func) {

  # wrapper function to filter signal panel of image
  kernel_filt <- function(x) {
    row_idx <- feat_map_idx[x, "Var1"]
    col_idx <- feat_map_idx[x, "Var2"]
    img_panel <- img[row_idx:(row_idx+kernel_dim[1] - 1), col_idx:(col_idx+kernel_dim[2] - 1)]
    filted_panel <- sum(img_panel * kernel)
    return(filted_panel)
  }
  # Apply complete convolution
  feat_map <- matrix(
    sapply(rownames(feat_map_idx), FUN = kernel_filt),
    nrow = dim_out,
    ncol = dim_out
  )
  activation_func(feat_map)
}

# pads kernel with zeros to be same dimensions as image, needed for fft convolution
pad_kernel <- function(kern, img_dim) {
  padding_len <- img_dim[1] - dim(kern)[1]
  right <- matrix(0, nrow = dim(kern)[1], ncol = padding_len)
  bottom <- matrix(0, nrow = padding_len, ncol = img_dim[2])
  padded <- rbind(cbind(kern, right), bottom)
  return(padded)
}

kernel_conv_fft <- function(img, img_dims, padded_ft, dim_out, activation_func) {
  convolved <- fft(fft(img) * padded_ft, inverse = TRUE)
  convolved_trunc <- suppressWarnings(as.numeric(
    convolved[dim_out[1]:img_dims[1], dim_out[2]:img_dims[2]]
  ))
  activation_func(convolved_trunc)
}

apply_kernel_conv <- function(
  imgs, kern, kernel_dim, stride, method = "fft", activation_func = relu
) {
  data_dim <- dim(imgs)
  dim_out <- ((data_dim[1] - kernel_dim[1]) / stride) + 1

  if (method == "fft") {
    # calc truncation of image to match slow method
    trunc_dims <- c(data_dim[1] - dim_out + 1, data_dim[2] - dim_out + 1)
    # zero pad kernel to match size of images
    padded <- pad_kernel(kern[kernel_dim[1]:1, kernel_dim[2]:1], data_dim[1:2])
    padded_ft <- fft(padded)
    conv_func <- function(x) {
      kernel_conv_fft(x, data_dim[1:2], padded_ft, trunc_dims, activation_func)
    }
  } else {
    # generate two column matrix of all indeces needed for full convolution
    conv_idx <- seq(1, data_dim[2] - kernel_dim[2] + 1, by = stride)
    feat_map_idx <- expand.grid(rep(list(conv_idx), 2))
    conv_func <- function(x) {
      kernel_conv_slow(x, kern, kernel_dim, feat_map_idx, dim_out, activation_func)
    }
  }
  feat_maps <- apply(imgs, MARGIN = 3, conv_func)
  feat_maps <- array(feat_maps, dim = c(dim_out, dim_out, data_dim[3]))
  return(feat_maps)

}
```

Below I have benchmarked the two convolution implementations. I also show the differences in the images after convolution.

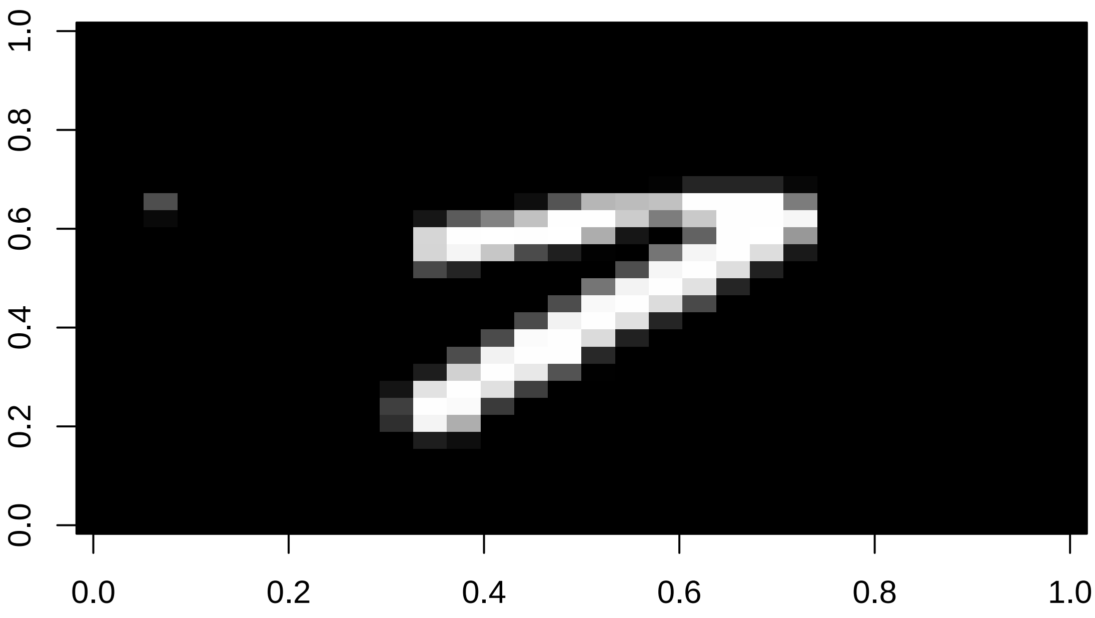
```
stride <- 1
kernel_dim <- c(5, 5)
set.seed(7734)
kern <- matrix(
  rnorm(prod(kernel_dim), 0, 1.5),
  nrow = kernel_dim[1],
  ncol = kernel_dim[2]
)

ti <- Sys.time()
feature_maps_slow <- apply_kernel_conv(
  train_dat[["imgs"]][, , 1:100],
  kern,
  kernel_dim,
  stride,
  method = "slow",
  activation_func = relu
)
tf <- Sys.time()
print(tf - ti)
```

```
## Time difference of 7.964366 secs
```

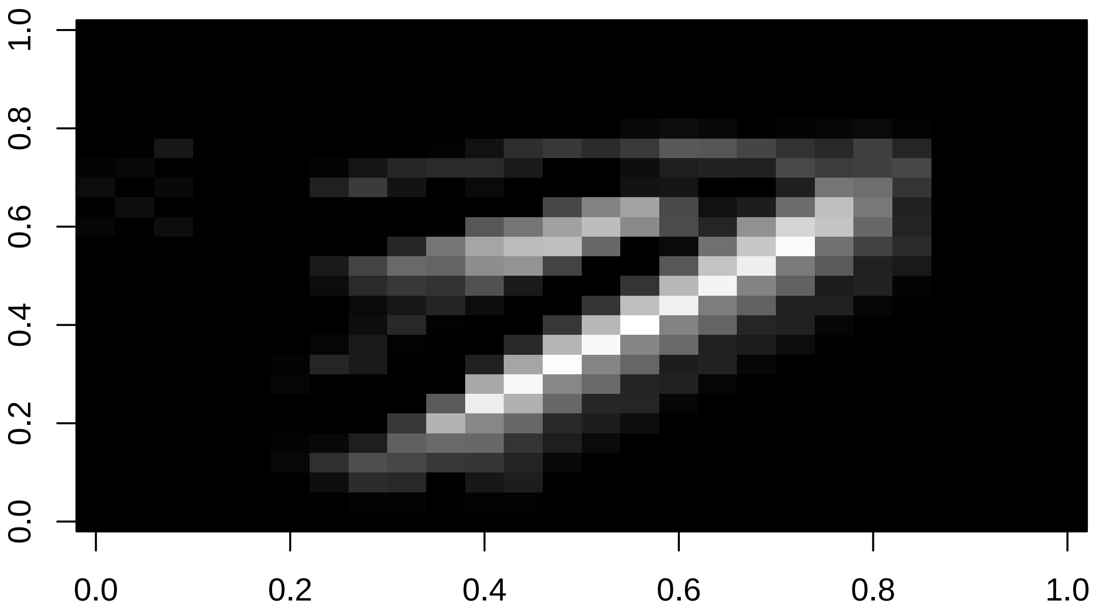
```
ti <- Sys.time()
feature_maps_fft <- apply_kernel_conv(
  train_dat[["imgs"]][, , 1:100],
  kern,
  kernel_dim,
  stride,
  method = "fft",
  activation_func = relu
)
tf <- Sys.time()
print(tf - ti)
```

```
## Time difference of 0.01353908 secs
```

```
## Raw Image
```



```
## image post kernel convolution in frequency domain
```



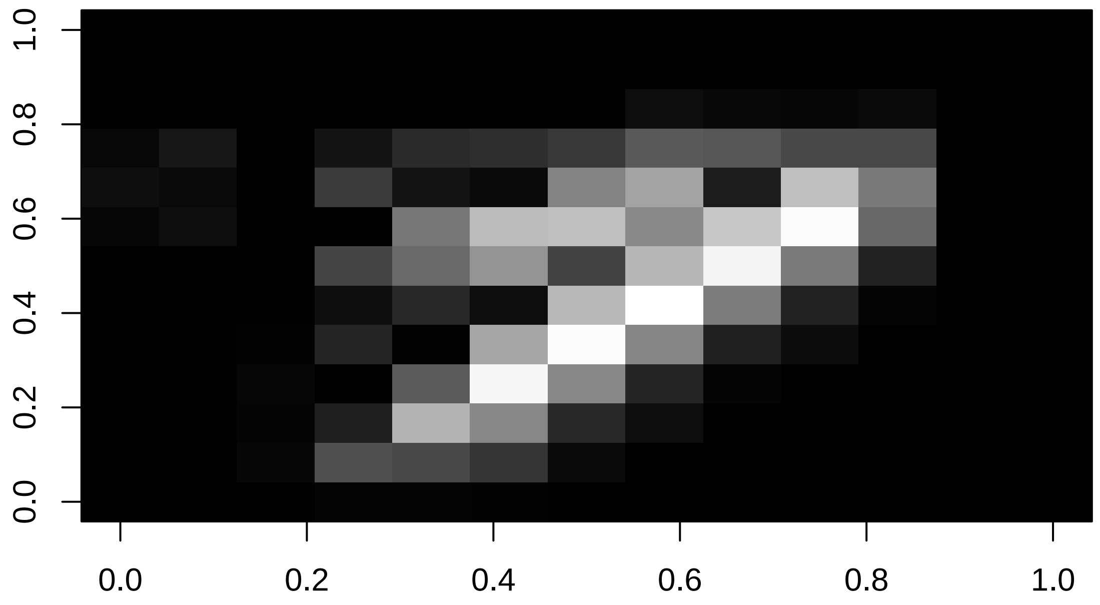
Pooling

Below I program some pooling procedures to implemnt a pooling layer within the convolutional neural network. I will re-use the code for slow convolution for the pooling operation.

```r
pooling_procedure <- function(img, filter_dim, feat_map_idx, dim_out, pooling_func = max) {

  # wrapper function to filter signal panel of image
  pool_filt <- function(x) {
    row_idx <- feat_map_idx[x, "Var1"]
    col_idx <- feat_map_idx[x, "Var2"]
    img_panel <- img[row_idx:(row_idx+filter_dim[1] - 1), col_idx:(col_idx+filter_dim[2] - 1)]
    filted_panel <- pooling_func(img_panel)
    return(filted_panel)
  }
  # Apply complete convolution
  pooled_map <- matrix(
    sapply(rownames(feat_map_idx), FUN = pool_filt),
    nrow = dim_out,
    ncol = dim_out
  )
  return(pooled_map)
}

pooling_layer <- function(imgs, filter_dim, stride, pooling_func = max) {

  data_dim <- dim(imgs)
  dim_out <- ((data_dim[1] - filter_dim[1]) / stride) + 1
  pool_idx <- seq(1, data_dim[2] - filter_dim[2] + 1, by = stride)
  pool_map_idx <- expand.grid(rep(list(pool_idx), 2))

  pool_func <- function(x) {
    pooling_procedure(x, filter_dim, pool_map_idx, dim_out, pooling_func = pooling_func)
  }

  dim_out <- c(dim_out, dim_out, data_dim[3])
  pooled <- apply(imgs, MARGIN = 3, pool_func)
  pooled <- array(as.numeric(pooled), dim = dim_out)

  return(pooled)
}
pool_stride = 2
ti <- Sys.time()
pooled_fft <- pooling_layer(feature_maps_fft, c(2, 2), pool_stride, pooling_func = max)
tf <- Sys.time()
print(tf - ti)
```

```
## Time difference of 0.6199579 secs
```

```r
image(pooled_fft[, , 30], col = gray((0:255) / 255))
```



```r
dense_layer <- function(feature_maps, weights, biases, activation_func) {
  node_vals <- weights %*% feature_maps + biases
  output <- activation_func(lin_transform)
}
```

Backward Propagation