UNIVERSITÀ DEGLI STUDI DI MILANO

Statistical Learning, Deep Learning and Artificial Intelligence

Instructor: Respected Prof. Silvia Salini

Student: Pramodh Chintharedddy (936461) - Data Science and Economics

Project: MNIST dataset Classification using Deep Learning

Index:

1.Abstract:

This report is an analysis on the famous MNIST dataset for handwritten digit recognition. This dataset has been extensively used to validate novel techniques in computer vision, and in recent years, many researchers have explored the performance of convolutional neural networks (CNNs) and other deep learning techniques over this dataset. The report describes the usage of neural networks for the classification task (Pattern recognition task). Explored variants of Neural Network Architectures such as Multi-layer Perceptron, Convolutional Neural Networks and Dropout and L2 regularization methods to study the performance for the digit recognition task.

2. The MNIST dataset, Problem Statement, Project Goal:

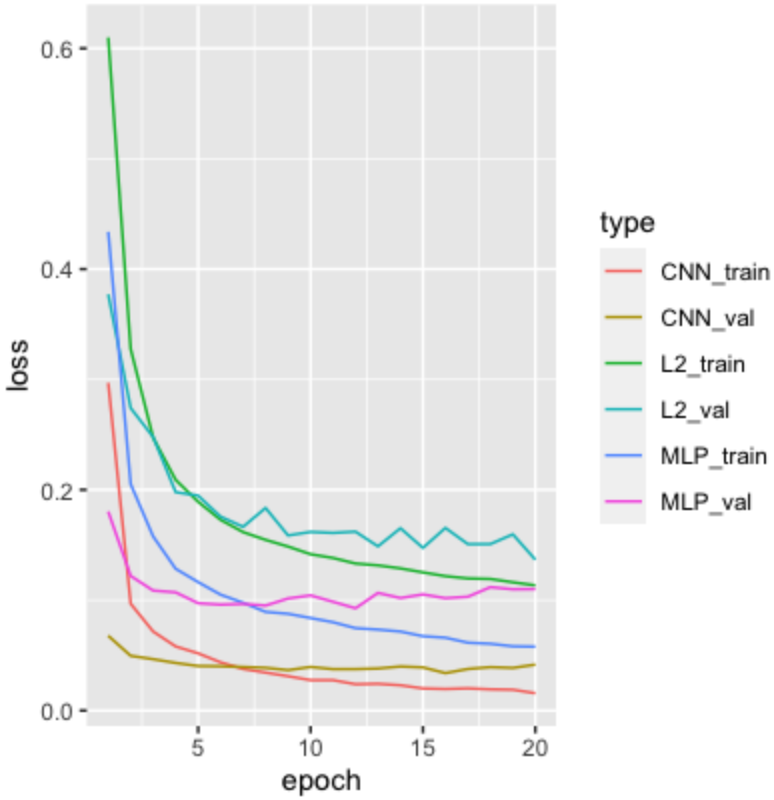
MNIST is a dataset of handwritten digits, with a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image. This is a good database to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

The images contain grey levels centered in a 28*28 field. Sets of writers of the training set and test set were disjoint.

Our goal of this project is the digit recognition task from the images using Neural Networks.

3. Key Findings:

- Neural networks are proven to be good algorithms at Image Classification tasks
- Dropout worked as a better regularization approach compared to Ridge (L2 regularization)
- Convolution Neural Network performs better than a Multi-Layer Perceptron
- Properties of CNN to retrieve input from Multidimensional inputs are an interesting approach to solve problems within the field of computer vision.
- Convolution operations are much more complex than dense layer (dense layers need only a fraction of time with respect to convolutional layers to run)



4. Analysis:

4.1 Using a Multi-layer Perceptron:

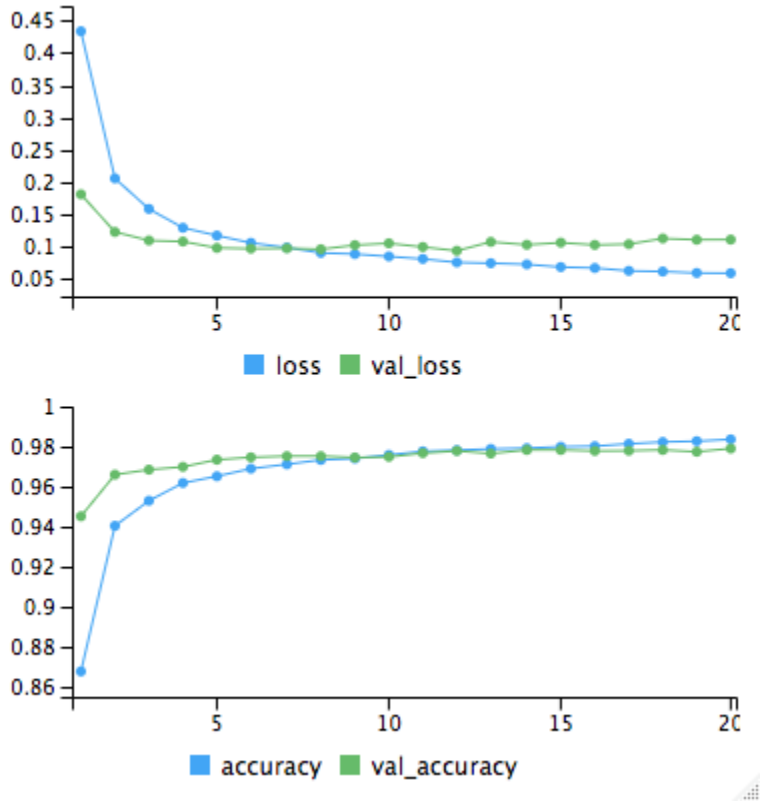
A model with 2 dense layers (each with 256 and 128 neurons respectively) using **relu** activation function and **dropout** (40% and 30%), with the final output layer containing 10 neurons (each corresponding to 1 different label) with a **SoftMax** activation.

SoftMax activation function for Classification because it creates an exponential probability distribution in the range of (0,1) making it easy for classification based on the activation with highest probability.

Loss function used for all the experiments: Categorical Cross Entropy, which makes the labels mutually exclusive for each training data point making it suitable for our **Single-label Classification**

Dropout as a Regularization method, helps in training a complex model without additional computational expense of training and maintaining multiple models, by randomly dropping out nodes during training.

Trained the model using **rmsprop** optimizer and **categorical_crossentropy** as loss function. With a batch size of 128 and 20% validation split, my model achieved a **val_accuracy** of **97.94%** when trained up to 20 epochs.



When evaluated on the test set achieved an accuracy of **98.09%**

4.2 Using Multi-layer Perceptron with Regularization:

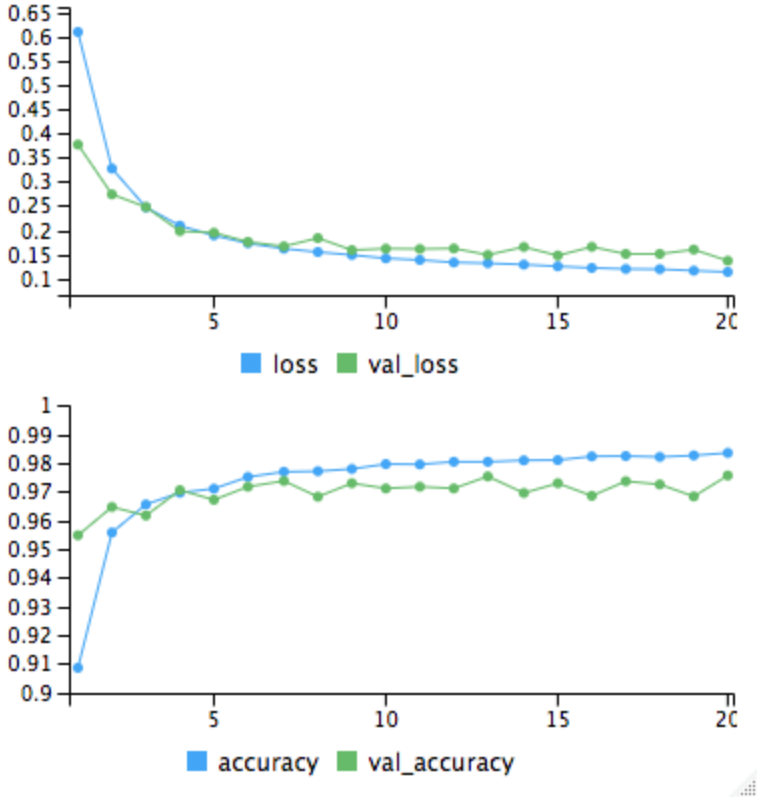
Used a **L2 regularizer(also called Ridge)**, whose penalty is computed as:

Loss = l2 * reduce_sum(square(x))

i.e. L2 regularization adds a penalty equal to the sum of the squared value of the coefficients.

The L2 regularization will force **the parameters to be relatively small**, the bigger the penalization, the smaller (and the more robust) the coefficients are.

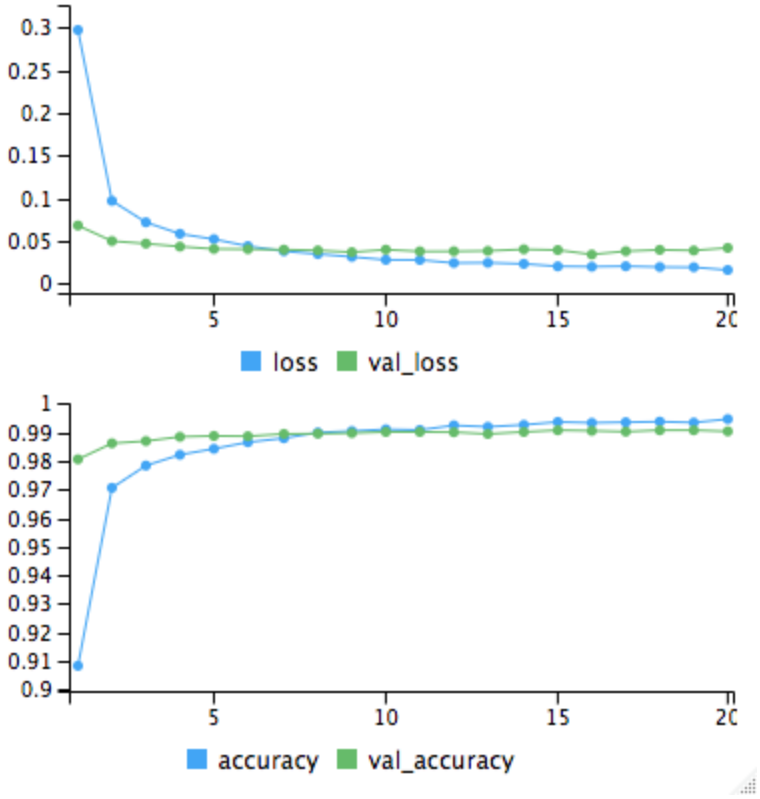
With a l2 value of **0.001**, and the same network architecture as our previous model but without dropout, achieved a **val_accuracy** of **97.57%**



When evaluated on the test set achieved an accuracy of **97.69%**

4.3 Using a Convolutional Neural Network:

In our final model using a CNN, with **2 Conv2D layers** (with 32 and 64 filters respectively), 1 **MaxPool**, and a **Dropout** of 25%, a dense layer of 128 neurons with 50% dropout before going to the final layer. Like our previous models we used the same activation and loss functions with **adadelta** optimizer for training. Upon training for 20 epochs with a batch size of 128 and 20% Validation Split to achieve a **val_accuracy** of **99.04%**

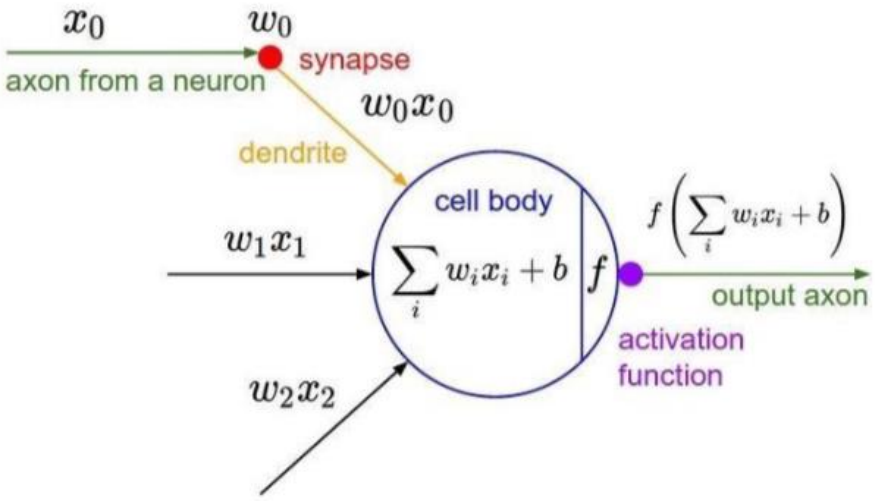


When evaluated on the test set achieved an accuracy of **99.11%**

5. Theoretical background of methods used:

5.1 Multi-layer Perceptron:

Neural networks are parametric algorithms that train on a predetermined set of variables (unlike a k-NN or Tree Predictors). Neural Networks are linear classifiers that try to differentiate the different data points using Separating Hyperplanes (In linearly separable case).



Neural Networks train with help of a Convex loss function where the goal is to find the local minima and descent in the negative direction of the gradient proportional to the gradient (A commonly known technique called Gradient descent – which happens inside the Backpropagation algorithm).

Algorithm: Perceptron
Input: Training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)$.
Initialization $\boldsymbol{w} = (0, \ldots, 0)$.
Repeat
  Read next training example $(\boldsymbol{x}_t, y_t)$
  If $y_t \boldsymbol{w}^\top \boldsymbol{x}_t \leq 0$, **then** $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_t \boldsymbol{x}_t$
Until $\gamma(\boldsymbol{w}) > 0$
Output $\boldsymbol{w}$

The Perceptron algorithm finds a homogeneous separating hyperplane by running through the training examples one after the other. The current linear classifier is tested on each training example and, in case of misclassification, the associated hyperplane is adjusted.

If the algorithm terminates, then w is a separating hyperplane. Perceptron always terminates on linearly separable training sets.

The Perceptron algorithm accesses training data in a sequential fashion, processing each training example in linear time, making Perceptron very competent on large training data sets. It is also very good at dealing scenarios in which new training data are generated all the time.

Parameters: Class $\mathcal{H}$ of predictors, loss function ℓ.

The algorithm outputs a default initial predictor h_1

For $t = 1, 2, \ldots$

1. The next example $(\boldsymbol{x}_t, y_t)$ is observed
2. The loss $\ell\big(h_t(\boldsymbol{x}_t), y_t\big)$ of the current predictor h_t is computed
3. The online learner updates h_t generating a new predictor h_{t+1}

Model update h(t) --> h(t+1) is typically local.

Projected OGD

Parameters: $\eta > 0, U > 0$

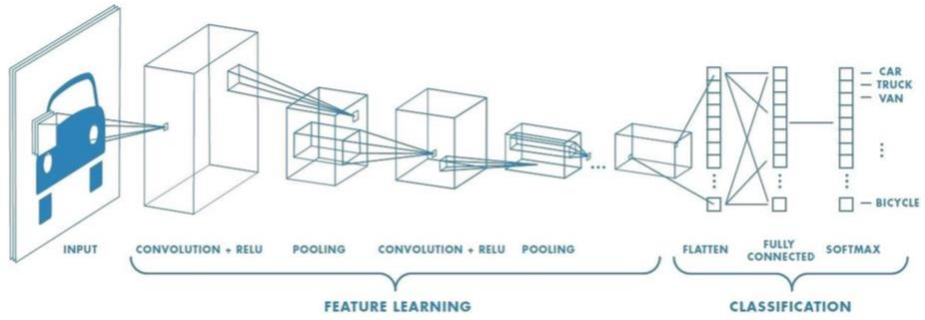
Initialization: $\boldsymbol{w}_1 = \boldsymbol{0}$

For $t = 1, 2, \ldots$

1. $\boldsymbol{w}'_{t+1} = \boldsymbol{w}_t - \dfrac{\eta}{\sqrt{t}} \nabla \ell_t(\boldsymbol{w}_t)$
2. $\boldsymbol{w}_{t+1} = \underset{\boldsymbol{w}\,:\,\|\boldsymbol{w}\| \le U}{\operatorname{argmin}} \left\|\boldsymbol{w} - \boldsymbol{w}'_{t+1}\right\|$

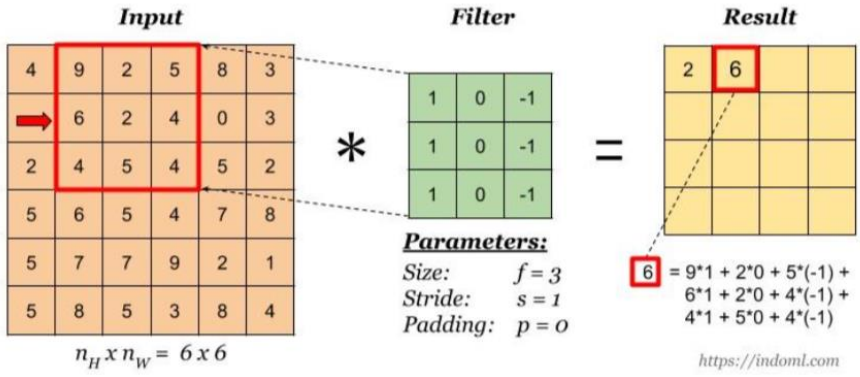
5.2 Convolutional Neural Network:

In Convolutional Neural Network, unlike Regular Neural Network connecting to all other neurons from the previous layer, while processing a part of an image it connects to the only required neurons which are in its surroundings or close to it. Also, CNN take advantage of the shared-weights architecture reducing the memory footprint of the network. CNN like any other Neural network learns through the process of Backpropagation.



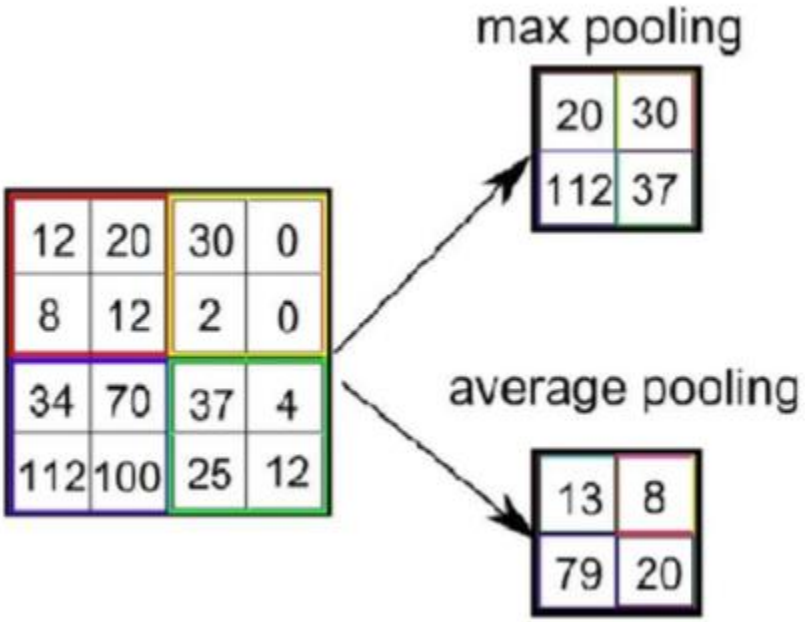
Convoluting - An image is composed on number of channels mentioned with the filter being applied on it, we can choose the number of filters. At end, we apply activation function; can be a Max function or ReLU function. As output, a tensor is created for each filter applied.

Again, these tensors are allowed as inputs to the next layer in the convolution. Then, number of filters has to be chosen and applied on the channels. Later, on applying activation function, the same Max function tensors are produced as output, equal to the number of filters applied earlier.



The total number of multiplications to calculate the result above is (4 x 4) x (3 x 3) = 144.

Pooling - It aggregates some values into a single output value. It reduces the size of the output tensor from each input tensor. Depth remains the same after pooling. *Max Pooling* takes the maximum input of a particular convoluted feature. *Average Pooling* takes the average input of a particular convoluted feature.



Fully Connected - Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional Multi-Layer Perceptron Neural Network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

Number of parameters in the network: Input * Output + Biases

5.3 Dropout:

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by **1 / (1 - rate)** such that the sum over all inputs is unchanged.

5.4 Regularization:

Regularization adds a penalty on the different parameters of the model to reduce the freedom of the model. Hence, the model will be **less likely to fit the noise** of the training data and will improve the generalization abilities of the model.

- The L1 regularization (also called Lasso)
- The L2 regularization (also called Ridge)
- The L1/L2 regularization (also called Elastic net)

Regularizers allow us to apply penalties on layer parameters or layer activity during optimization. These penalties are summed into the loss function that the network optimizes.

Regularization penalties are applied on a per-layer basis. The exact API will depend on the layer, but many layers (e.g. Dense, Conv1D, Conv2D and Conv3D) have a unified API.

These layers expose 3 keyword arguments:

- **Kernel_regularizer**: Regularizer to apply a penalty on the layer's kernel
- **Bias_regularizer**: Regularizer to apply a penalty on the layer's bias
- **Activity_regularizer**: Regularizer to apply a penalty on the layer's output

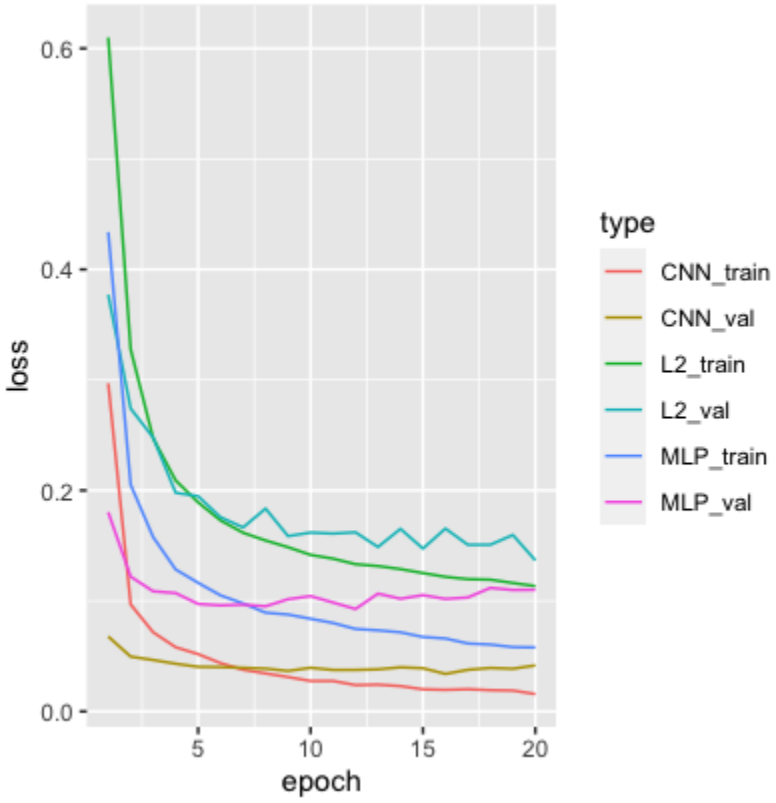
The value returned by the activity_regularizer object gets divided by the input batch size so that the relative weighting between the weight regularizers and the activity regularizers does not change with the batch size.

We can access a layer's regularization penalties by calling **layer.losses** after calling the layer on inputs.

6. Conclusions:

- Neural networks are proven to be good algorithms at Image Classification tasks
- Dropout worked as a better regularization approach compared to Ridge (L2 regularization)

- Convolution Neural Network performs better than a Multi-Layer Perceptron
- Properties of CNN to retrieve input from Multidimensional inputs are an interesting approach to solve problems within the field of computer vision.
- Convolution operations are much more complex than dense layer (dense layers need only a fraction of time with respect to convolutional layers to run)



7. Appendix:

R Code:

```
## MNIST Dataset Classification using Deep Learning:


#1 - Loading required libraries:

devtools::install_github("rstudio/keras")  #skip if already
installed

library(keras)

install_keras()

library(tensorflow)
```

```
install_tensorflow(version = "nightly") #skip if already
installed

library(keras)


#2 - Loading the Dataset:

mnist <- dataset_mnist()

x_train <- mnist$train$x

str(x_train)

y_train <- mnist$train$y

x_test <- mnist$test$x

y_test <- mnist$test$y


dim(x_train)


#3 - Preprocessing:

  # Our x data is in a 3-d array (images,width,height)

  # Converting the 3-d arrays into matrices by reshaping width
and height into a single dimension for training a MLP

  # reshape: 28*28 = 784 (28x28 images are flattened into length
784 vectors)


#Reshaping array to CNN 3D Input_Shape format by adding channel
i.e, (width,height,channel):

X_train_CNN <- array_reshape(x_train, c(nrow(x_train), 28, 28,
1))

X_test_CNN <- array_reshape(x_test, c(nrow(x_test), 28, 28, 1))

str(X_train_CNN)


str(x_train)
```

```
#Reshaping array to MLP Input Format:
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
str(x_train)
x_test <- array_reshape(x_test, c(nrow(x_test), 784))

# Converting the grayscale values from integers ranging between
0 to 255 into floating point values
# ranging between 0 and 1 (Rescaling):
x_train <- x_train / 255
x_test <- x_test / 255

X_train_CNN <- X_train_CNN / 255
X_test_CNN <- X_test_CNN / 255

# y data contains labels from 0 to 9. For training we one-hot
encode the vectors into binary class matrices using the
# Keras to_categorical() function (hence we will use the
categorical loss function):
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)



#4 - Defining the Network Architecture:
#Using Multi-layer Perceptron:
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu',
              input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
```

```
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')

# The input_shape argument to the first layer specifies
# the shape of the input data (a length 784 numeric vector
# representing a grayscale image). The final layer
# outputs a length 10 numeric vector (probabilities for
# each digit) using a softmax activation function.

summary(model)

# compiling the model
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

# fitting the model in batches of 128 images and 30 epochs
history <- model %>% fit(
  x_train, y_train,
  epochs = 20, batch_size = 128,
  validation_split = 0.2
)

plot(history)

# evaluate accuracy
```

```
model %>% evaluate(x_test, y_test)

# prediction
model %>% predict_classes(x_test)



# MLP using regularization:
modelReg <- keras_model_sequential()

modelReg %>%
  layer_dense(units = 256, activation = 'relu',
              input_shape = c(784),
              kernel_regularizer = regularizer_l2(l = 0.001))
%>%
  layer_dense(units = 128, activation = 'relu',
              kernel_regularizer = regularizer_l2(l = 0.001))
%>%
  layer_dense(units = 10, activation = 'softmax')

summary(modelReg)

modelReg %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

historyReg <- modelReg %>% fit(
  x_train, y_train,
```

```
  epochs = 20, batch_size = 128,
  validation_split = 0.2
)


# evaluate accuracy
modelReg %>% evaluate(x_test, y_test)

# prediction
modelReg %>% predict_classes(x_test)

#Using CNN:
modelCNN <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation =
"relu",
                input_shape = c(28, 28, 1)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation =
"relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = "softmax")

summary(modelCNN)

modelCNN %>% compile(
  loss = loss_categorical_crossentropy,
```

```
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)


historyCNN <- modelCNN %>%
  fit(
    X_train_CNN, y_train,
    epochs = 20,
    batch_size = 128, validation_split = 0.2
  )

# evaluate accuracy
modelCNN %>% evaluate(X_test_CNN, y_test)

# prediction
modelCNN %>% predict_classes(X_test_CNN)

#Comparing the 3 models:

library(tidyr)
library(tibble)
library(dplyr)
library(ggplot2)

compare_cx <- data.frame(
  MLP_train = history$metrics$loss,
  MLP_val = history$metrics$val_loss,
```

```
  L2_train = historyReg$metrics$loss,
  L2_val = historyReg$metrics$val_loss,
  CNN_train = historyCNN$metrics$loss,
  CNN_val = historyCNN$metrics$val_loss
) %>%
  rownames_to_column() %>%
  mutate(rowname = as.integer(rowname)) %>%
  gather(key = "type", value = "value", -rowname)

ggplot(compare_cx, aes(x = rowname, y = value, color = type)) +
  geom_line() +
  xlab("epoch") +
  ylab("loss")
```

8. References:

[1]: http://cesa-bianchi.di.unimi.it/MSA/Notes/linear.pdf

[2]: http://cesa-bianchi.di.unimi.it/MSA/Notes/online.pdf

[3]: https://keras.io/api/layers/regularization_layers/dropout/

[4]: https://keras.io/api/layers/regularizers/

[5]: https://www.r-bloggers.com/machine-learning-explained-regularization/

[6]: https://en.wikipedia.org/wiki/Convolutional_neural_network

[7]: https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/