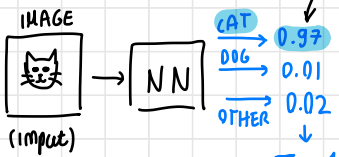
GETTING STARTED WITH NN

NN: What they are, and why they matter

↓ simplify details to grasp MOST basic concept

- NN as a black-box — highest PROB is class CAT → "The NET predict the image as CAT"



outputs $\in [0,1]$, each one is the probability that input image is CAT, DOG, OTHER

class labels (category)

$\Sigma_i = 1$ being prob

this is an image classification problem where input "image" → output "numerical value ∀ class (NOT labels itself)" ← the chosen predicted label is the one with highest probability

even if $P_{CAT} = 0.51$, $P_{DOG} = 0.48$, $P_{other} = 0.01$ → still CAT, but less confident about prediction

a perfect NN output $P_{CAT} = 1$, $P_{DOG} = 0$, $P_{other} = 0$ when INPUT = cat image

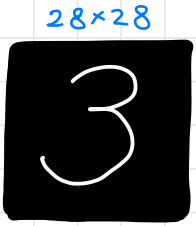
↳ in reality even well trained NETWORK doesn't give such result!

how all these elements on the predicting steps are represented?

INPUT (image) → NN (Network Model) → OUTPUT PREDICTION (probability distribution)

INPUT → how it can be represented as a numeric value?

GRAY SCALE images are represented as an array of pixel values where each pixel has intensity 0 ÷ 255 (black ÷ white)



Grayscal Image

→ [[0 00 ... 1 12 0 0 ...0],
[0 - - - 50 253 253 0..0],
⋮
[0 ... 0.. - - - - .0]]

28×28 matrix $M_{ij} \in [0,255]$

⇓

Similarly color images has 3 components 28×28×3 matrix of RGB intensity levels!

28×28×3 → numbers represent a colored 28×28 pixel image!

total amount of numbers as input to the network

⇒ HOW IT IS REPRESENTED?
It depends on the network..

- 1 dimensional Vector input $\in \mathbb{R}^{28 \cdot 28 \cdot 3}$
- three 2D arrays each 28×28 input $\in \mathbb{R}^{28 \cdot 28 \times 3}$

the Network design expect a fixed size and shape for data

When the image is NOT coherent with the expected input...
↳ we resize / crop to the expected size

IF GRAYSCACE we can replicate it for all 3 channels (when RGB expected)
or opposite when grayscale is expected RGB is converted
↓
PRE-PROCESSING step to create suitable input

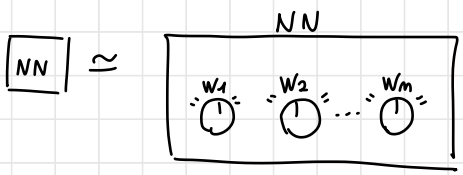
NN are designed to accept a certain shape / size INPUT

↓

different image classification networks use different INPUT based on the application they are designed for!

"TRAIN a NEURAL NETWORK"

↳



contains many tunable parameters ~ KNOBS inside NN

↑

weights

⇓

← Network weights $w_1, w_2, \ldots, w_m$

essential to use the NN...

you find the correct weights by TRAINING the NN

↓

showing thousands of examples of classes you want to learn

SUPERVISED LEARNING

↳ you provide the NN with image of a class and you explicitly tell the result

⇓

IF the network model makes a WRONG prediction, we compute the error related to the bad prediction, and use it to adjust network weights → so that subsequent prediction accuracy are improved

↳ details on the training step are discussed next..

FUNDAMENTALS OF TRAINING A NEURAL NETWORK

most essential elements required for TRAINING a NN
- Label traiming data
- loss function
- Updating weights

example specific for an IMAGE CLASSIFICATION NETWORK TRAINING

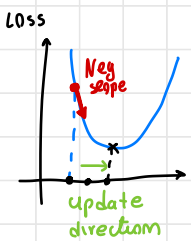
⇓

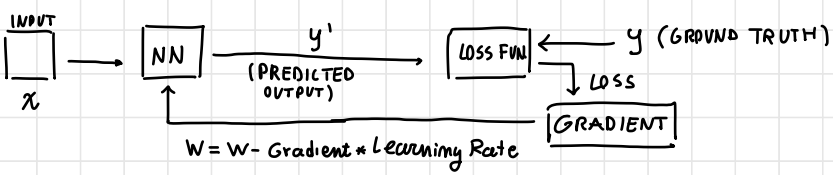
the TRAINING requires

1) Labeled traiming data = thousands of images ∀ class with expected output (Label)

2) Loss Function ≃ COST FUNCTION: numerical computation that quantify error between network output and expected result
here we rely on MEAN SQUARED ERROR

3) Process for tuning the NN's weights based on the value of the LOSS FUNCTION (GRADIENT DESCENT)





1 and 2: Representation of traiming data and loss function computation

1. TRAINING DATA:

Input image (x) (numerical matrix)

"One-Hot Encoding" → is recommended when classes are NOT related.. representing categorical labels as BINARY VECTORS

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ class labels (numerically)

"CAT" label

"DOG" would be $y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

← One-Hot Encoding used to represent class labels

↖ this known labels are called "GROUND TRUTH" := y

x (INPUT image) → NN → CAT / DOG / OTHER

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ GROUND TRUTH

$y' = \begin{bmatrix} 0.37 \\ 0.50 \\ 0.13 \end{bmatrix}$ (OUTPUT) PREDICTED VALUE

this would predict DOG as class WRONG!

← estimate of the GROUND TRUTH

⇓

to quantify the ERROR between the Network output (PREDICTION) and the RESULT EXPECTED is to compute the sum of squared error

$y \in \mathbb{R}^N, y' \in \mathbb{R}^N \quad SSE = \sum_{j=1}^{N} (y[j] - y'[j])^2$ ← error for a single training sample using prediction and ground truth

↓

When training a NN MANY (multiple) images are used to compute the loss before updating Network weights

↘ we use Mean Squared Error for multiple (m) training images ← refered as BATCH SIZE (32 typical)

each batch of image is defined an ITERATION

$MSE = \text{mean}\{SSE\} = \frac{1}{m} \sum_{i=1}^{m} (y_i - y_i')^2$

other loss functions can be used requiring a numerical representation for input class and prediction (y and y')

representation of error between expected output and input to Network

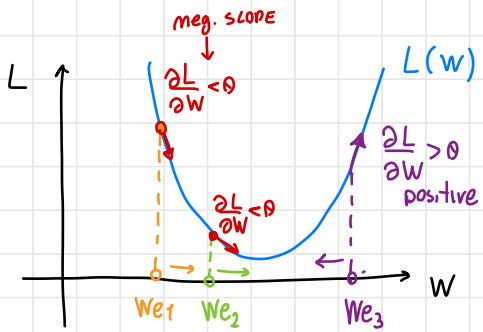
↓

3. How can we use this loss information to train the Network?

this TUNING of weights of NN can be achieved by GRADIENT DESCENT Algorithm

GRADIENT DESCENT (Gradient ~ SLOPE)

example of this Algorithm with only W (one parameter) and loss L convex



$\Rightarrow$ the GRADIENT sign $\frac{\partial L}{\partial W}$ is what matters!

↓

we want L to decrease so based on the fact that

$\text{Slope} = \frac{\text{rise} \uparrow}{\text{run} \leftrightarrow} < 0$ when moving towards minimum!

↓

we need to adjust the weights in the direction toward decreasing loss

IF $\frac{\partial L}{\partial W} > 0$ we move left ←

IF $\frac{\partial L}{\partial W} < 0$ we move right →

⇓

we need to adjust the weights in the direction opposite to the sign of the gradient

$$We_{t+1} = We_t - \text{GRADIENT} * \text{LEARNING Rate}$$

the sign of gradient determine the direction of motion

the amount of motion (step size) depend on the Learning Rate parameter < 1 it is a parameter we initialize (NOT determined by the NET) that determines step

this is an HYPERPARAMETER ↑ because it is different from trainable parameters such as Net weights

← equation to update the weight in the proper direction regardless of current W relative to optimal W^*

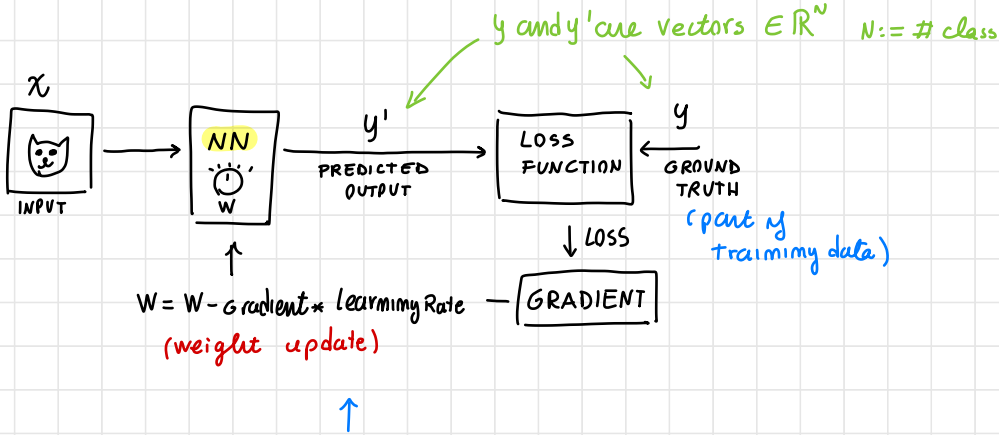
In practice $L(W_1 \dots W_N)$ has many dimension, NOT convex, many pick/valley

Its slope is called GRADIENT and it is function of all Network weights

↳ so GRADIENT is a VECTOR in multi dimensional space...
(but the previous approach is used)

this GRADIENT of LOSS FUNCTION with respect to network weights is computed with an Algorithm BACKPROPAGATION
(built in on DL FrameWorks as TensorFlow OR Pytorch)

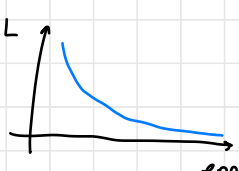
NN TRAINING PROCESS



Training a NN is an ITERATIVE PROCESS, often requires passing entire TRAINING set through the NET Multiple times!

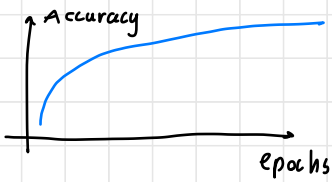
↳ each time the TRAINING SET is passed through the Net is define as TRAINING EPOCH

training a NN often Requires many training epochs until Loss decrease satisfactory



↝ training progresses until Loss Function stabilize and STOP decreasing

similarly ACCURACY is evaluated



⇒ many advanced topics on NN training are neglected here... such as DATA SPLITTING

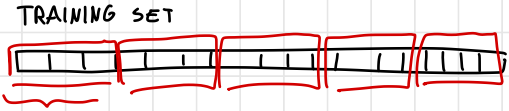
What you do with your TRAINED NN ?

using the NETWORK to perform inference

you can then feed the NN with new images x and predict y' on NON CLASSIFIED images

↑ without a label training data

- Remember that after INPUT must be PRE-PROCESSED to be coherent with expected NN input
- NN weights are initialized to small random values
- BATCH, ITERATIONS and EPOCHS



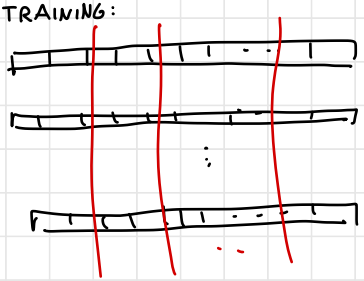
M labeled training data

B := batch size → ITERATION

iterations $\forall$ epoch e

iterations = I = M/B

↳ TRAINING:



training epoch 1

training epoch 2

training epoch E

LINEAR REGRESSION (LR)

how simple NN can be used to solve LR problem using Tensorflow and Keras

↓ it make easy to load the dataset, define NN and train it

LR, fit a straight line through a set of data points

↳ the power of NN comes from the ability to model highly non linear functions → here we use linear function to define basic terms

→ set up a Random Number SEED so that results are repeatable

1) Load dataset: boston housing dataset available in Keras (simple example)

↓

we get training and test dataset (*) with input features and target variables

{ x : input
y : output target ← here it is a number

↓

this is the medium price of the house for a given features

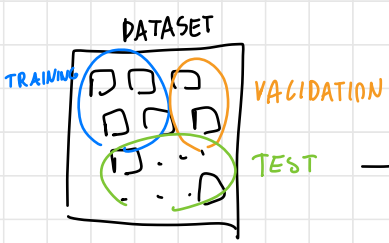
→ explore the dataset

.shape → (404, 13)

↑ training samples

↑ attributes = FEATURES

(*) DATA SPLITTING Data set { training, validation, test } ⇒ help prevent overfitting (when model doesn't generalize unseen data)



→ IF validation dataset NOT defined,

it is extracted from the TRAINING dataset

DATA SPLITTING IS fundamental in ML!

HERE we use simple features to keep simple + plot the Results

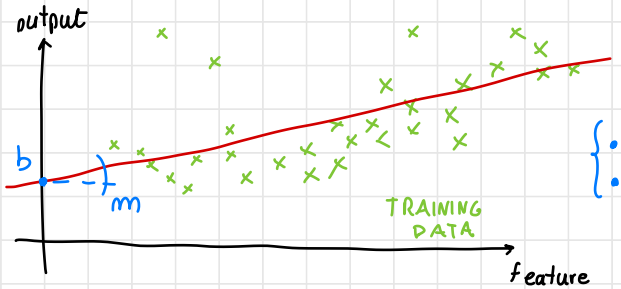
BUT it is important in general that many features are used to predict a single target value

→ 2) EXTRACT the 6TH feature only from dataset (here the highly correlated)

↓

good practice to visualize dataset, PLOT a portion of it to check if a reasonable feature has been chosen for prediction

Inspect data to be aware of Anomalies



} we try to obtain the model

- Y-intercept: b
- slope : m

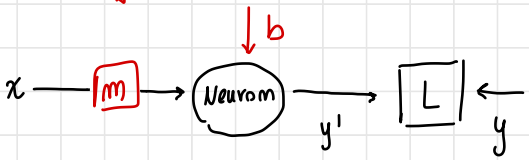
↑

best values of b, m that defines best FIT lines through this data

How to use NN to create such a model

In this problem

(m, b) are the weights of NN model



here we have one Neuron in a single layer

In general a NN has many Neurons arranged in Layers...

↓

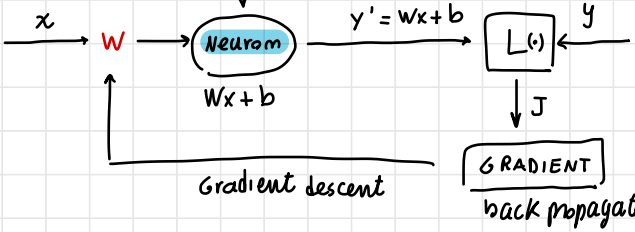
so DNN has many layers and many Neurons per layer

here we have two tunable parameters

- W: WEIGHT
- b: BIAS

↓

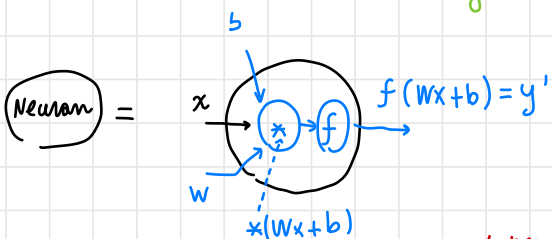
we wanna train the model to get the optimum w, b



In complex networks
with many W and b, the weight w_s are ~ function slope
b_s are ~ bias offset

• Neuron
"
it is a computation unit (PERCEPTRON) → computing $Wx+b$

then $Wx+b$ ~ is passed to an ACTIVATION FUNCTION f to produce the output of the neuron



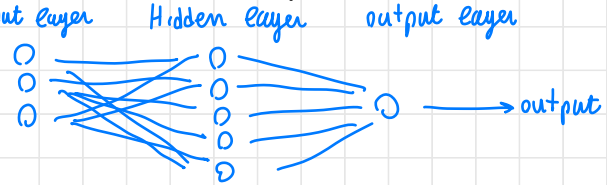
↳ in a LR we don't need an activation function
$f(\tilde{x}) = \tilde{x}$

when $f(\tilde{x})$ is specified, $\tilde{x}$ is floating point number and $y' = f(\tilde{x})$ is also a floating point number

⇓

Network like this are
SINGLE-LAYER PERCEPTRON (SLP)
(here even special case, because it has JUST one Neuron)

Network with multiple layers are MULTI-LAYER PERCEPTRON (MLP)

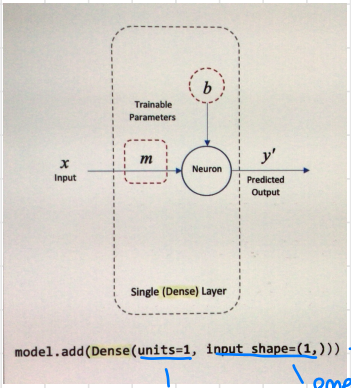


In our special SLP, we use y' and y to compute MSE (loss)
than GRADIENT to update weights ← (USING GRADIENT DESCENT)
↑ (managed by KERAS with BACKPROPAGATION) ← hidden in keras

$$J = \frac{1}{m} \sum_{i=1}^{m} (y'_i - y_i)^2$$

- code implementation:
  1) Define and compile the model
  2) Train model with fit
  3) use predict to make predictions

(1) instantiate the model = sequential()
than define it by model.add(...)
↑ with a single neuron



Dense (layer): every input is connected to every output (here one neuron)

the output is by definition a single number

one neuron — one input

than model.summary()
summarize the net architecture

- compile the model : prepare it for training ...

model.compile(...)
↑ specify the type of OPTIMIZER used to perform gradient descent and the type of loss function

(2) TRAIN the model with fit(·) method

It execute the training process := iterative cycle that updates the network weights

```
fit(x_train, y_train, batch_size, epochs, validation_split)
```

- **x_train**: features training data
- **y_train**: target value (ground truth)
- **batch_size**: we use a portion of training data to update... ~16 := 16 samples before update
  - batch size is an HYPERPARAMETER because it is preliminary specified, but it is important and can affect the model accuracy
  - [16 ÷ 256] depend on memory
  - 32 usually good start
- **epochs**: number of epochs we want to train model..
- **validation_split**: validation dataset, instruct to hold a percentage of data used for validation

⇓

this output the TRAINING EPOCHS for each time...
each epochs has #iterations = train# / batch size ↓ update
also train and validation
loss are returned by fit(·) method

history = model.fit(...)

↓

this history object allow to plot training validation Results

history.history['...'] dictionaries containing training / validation loss

reasonably $J_{TRAIN} < J_{VAL}$ because validation is done on data never seen before

9 we can use the model.predict(x) to make predictions over a list of x values

↓

We can plot the best fit line overlayed on training data to check accuracy of our LR model

⇓

how to evaluate the accuracy? → compute meaning full metrics on test data to determine if predictions within acceptable limit

for example the
RMS (Root Mean squared)
MAE (Mean Absolute Error)
from best fit on test dataset
(MSE is harder to interpret because it involve squared of price...)

When evaluating a model, use a metric easy to interpret! ←

↓

the LOSS FUNCTION used for OPTIMIZATION is NOT necessarily the same of performance metric!

↑

this metric are used for different purpose!

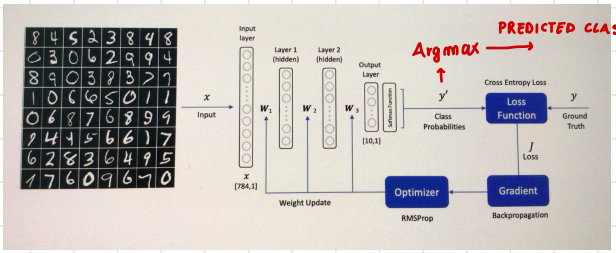
from the validation - prediction plot we can see big errors ... then maybe we should consider using additional features to improve predictions

With 2 input features the model became a plane
while hyperplane with more than 2 → still LINEAR MODEL

otherwise, we can consider to use non linear model!

MNIST DIGIT CLASSIFICATION USING MLP

How to implement a Feed Forward NN in Keras to classify hand written digits from MNIST Dataset

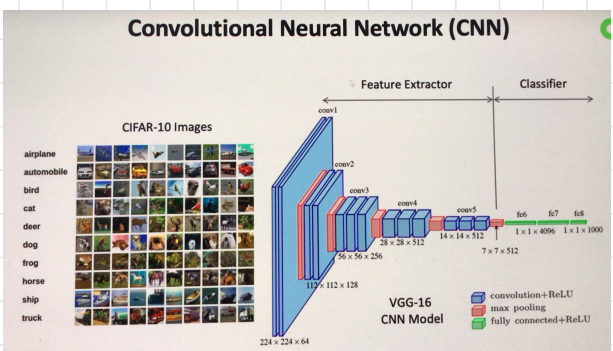


concept in the context of image classification involving more than two classes

⇓

Multinomial Regression (soft max Regression) → more than 2 classes

In general, MLP are NOT the best way to process image data BUT it is an example before covering advanced Network CONVOLUTIONAL NN (CNN) better suited for image processing (Addressed later)



MNIST dataset is included in Keras → easy to load

⇓

70000 images: partitioned { 60 000 training → We reserve a portion for validation (10000)
10 000 testing

{ 50 000 training
10 000 validation

28×28 pixels Images (GRAYSCALE)

1) LOAD and SPLIT Dataset

2) PREPARE dataset (PRE-PROCESS) for processing through Network by transforming

↳ here, the single input features (IMAGES)
logically it is represented by pixel intensity as features
↓
flatten the 2D image (28×28) data to 1D vector
Reshaping sample images on each set

+ Normalize pixel intensity by 255 to be $I_{xy} \in [0,1]$
(common when working with img data → help model to train efficiently)

How Image labels are represented? In categorical Data the target labels (STRING) must be converted to numerical values

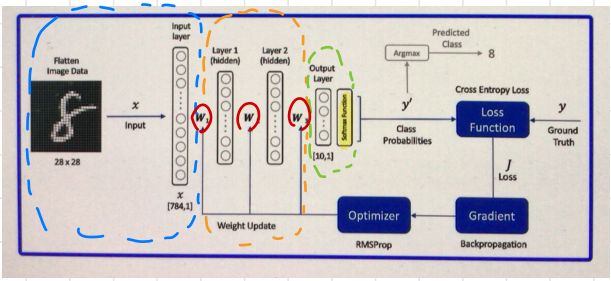
STRING → Numerical values : "Label Encoding" {
Option 1: ORDINAL INTEGER ENCODING (number int ∀ class)
Option 2: ONE-HOT ENCODING (separate binary vector to encode each class label)

Option 1: also mapping from class name to int representation is reported in a config file of dataset
→ unique integer are used to encode class labels
when class labels are NOT related one another,
⇓
Option 2 pre-processing, each labels is converted in N dimensional vector (with N as # classes)
↓
transform easily to one-hot encoding by to_categorical(·)

3) DEFINE Network architecture

Multiple layers:

- 1 INPUT LAYER
- 2 HIDDEN LAYERS
- 1 OUTPUT LAYER



3.1 Deep Neural Network Architecture

The network architecture shown below has multiple layers. An input layer, two hidden layers, and an output layer. There are several things to note about this architecture.

1. **Input Data:** The image input data is pre-processed (flattened) from a 2-Dimensional array `[28x28]` to 1-Dimensional vector of length `[784x1]` where the elements in this input vector are the normalized pixel intensities. The input to the network is sometimes referred to as the input "layer", but it's not technically a layer in the network because there are no trainable parameters associated with it.
2. **Hidden Layers:** We have two hidden layers that contain some number of neurons (that we need to specify). Each of the neurons in these layers has a non-linear activation function (e.g., ReLU, Sigmoid, etc...).
3. **Output Layer:** We now have ten neurons in the output layer to represent the ten different classes (digits: 0 to 9), instead of a single neuron as in the regression example.
4. **Dense Layers:** All the layers in the network are fully connected, meaning that each neuron in a given layer is fully connected (or dense) to each of the neurons in the previous layer. The **weights** associated with each layer are represented in bold to indicate that these are matrices that contain each of the weights for all the connections between adjacent layers in the network.
5. **Softmax Function:** The values from each of the neurons in the output layer are pased through a **softmax** function to produce a probability score for each of the ten digits in the dataset.
6. **Network Output:** The network output (y'), is a vector of length ten, that contains the probabilities of each output neuron. Predicting the class label simply requires passing (y') through the `argmax` function to determine the index of the predicted label.
7. **Loss Function:** The loss function used is **Cross Entropy Loss**, which is generally the preferred loss function for classification problems. It is computed from the ground truth labels (y) and the output probabilities of the network (y'). Note that y and y' are both vectors whose length is equal to the number of classes.

Although the diagram looks quite a bit different from the single-layer perceptron in the linear regression example, it is fundamentally very similar in terms of the processing that takes place during training and prediction. We still compute a loss based on the predicted output of the network and the ground truth label of the inputs. Backpropagation is used to compute the gradient of the loss with respect to the weights in the network. An optimizer (which implements gradient descent) is used to update the weights in the neural network.

- the INPUT image is transformed from 28×28 2D vector to 1D 784 vector with pixel intensity
  ↑
  INPUT LAYER, even if there are NO trainable parameters associated with that (it is NOT a real NETWORK layer)

the Hidden Layers contains neurons to specify:
↓
each neuron has a Non Linear Activation function such as ReLU / sigmoid

the Output Layer has N neurons (as # classes)
↑
here all layers are fully connected (DENSE) because each neuron in next layer is fully connected to previous layer

the weights W_i are matrix (∀ layer) containing the weight for cell connection between adjacent layers

Softmax function is used to the values of all output layers neuron ...
↓
to produce a probability score for each of the 10 digits in dataset
≃ it is a way of normalizing a set of inputs

Softmax: Output layer pass through softmax function to normalize them into probabilities

⇓

than $y' = \begin{bmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_N \end{bmatrix}$ each y'_j contain the probability of each output neuron (class j)

⇓

than the argmax is used to predict the label

As LOSS FUNCTION a CROSS-ENTROPY LOSS is used for classification problems → computed from y, y' (vectors of lenght = # classes)

↑

even if different to SLP, in MLP the processing to update the weights is the same!

4) MODEL IMPLEMENTATION (in Keras)

↓

Instantiate as Sequential() than use .add(...) to Build the architecture by 2 hidden layers + 1 output layers

'relu' ↑ 'softmax' ↓

the first hidden layer require the input shape + specify # Neurons as design choice (each neuron has its own activation function!)

here in output layers, the activation function Softmax is used to pass all the outputs through it!

~ 118.282 trainable parameters SMALL NET!
usually millions of parameters

5) COMPILE MODEL

specify
- type of optimizer (rmsprop)
- loss function (categorical_crossentropy) ← specify the label encoding IF integer encoded you use sparse_crossentropy
- additional metrics to track during training process [accuracy] training/validation loss automatically saved

L:= CROSS ENTROPY → simple computation that allow to quantify error between output probabilities and one-hot encoded vector ∀ class

6) TRAIN the MODEL by fit(...) method

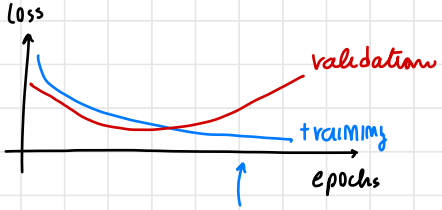
train dataset, epochs, batch_size, validation_data ← specify explicitly with separate validation dataset

return training_results history of training data

⇓

7) ANALYZE training Results

after first epochs... while training loss decrease, validation loss can increase when OVERFITTING is occuring!



training loss goes to 0, it is overfitting the data

common problem in machine learning, several techniques can mitigate overfitting issues...

↓

still accuracy ~ 98% is still very good... on average 1 mistake every 50 samples!

8) MODEL EVOCUATION

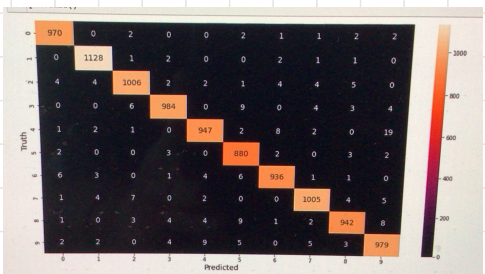
1. we can use the trained model to obtained predictions
   for any number of samples
   ↓
   we can apply predict to all the test data
   ↓
   containing the probability for each of the N (# class) neurons in the output layer

↓

an effecent way (INFORMATIVE) to evaluate the model on the entire Test set → is by CONFUSION MATRIX

↓

common metric used to summarize results of a classification problem



↓ this is presented as a table (matrix) where one axis is ground truth of each class while other axis is the prediction from model

↓

total # of entries are the # of instances from experiment

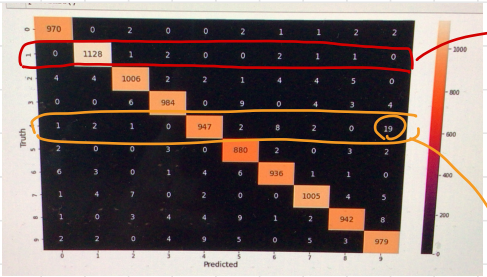
to evaluate it,
we extract for all test dataset the arg max prediction
since labels are one-hot encoded → we convert the prediction back to integer

than the confusion matrix is generated easily passing list of ground truth labels and predicted

↓

than we can plot this as heatmap

(or also provided as simple table of numbers or percentage)



the row 1 is the digit 1 ground truth..
here we get that we have
1128 correct predictions
and 7 samples wrong

WORST classification
because 4 ~ 9 similar shape

↓

Overall accuracy of 98% on validation...
from confusion matrix we can learn which
class is more difficult to predict → INSIGHT!

⇑

NN involves millions parameters in NON-visualizable dimensions...
modeling complex NON linear relationship

↓

focus on understanding the high level components
of a Neural Network!

↓

- GRADIENT DESCENT
- LOSS FUNCTION
- BACKPROPAGATION
- NN ARCHITECTURE
- ....

well established
algorithms
embedded in
all modern DNN
frameworks