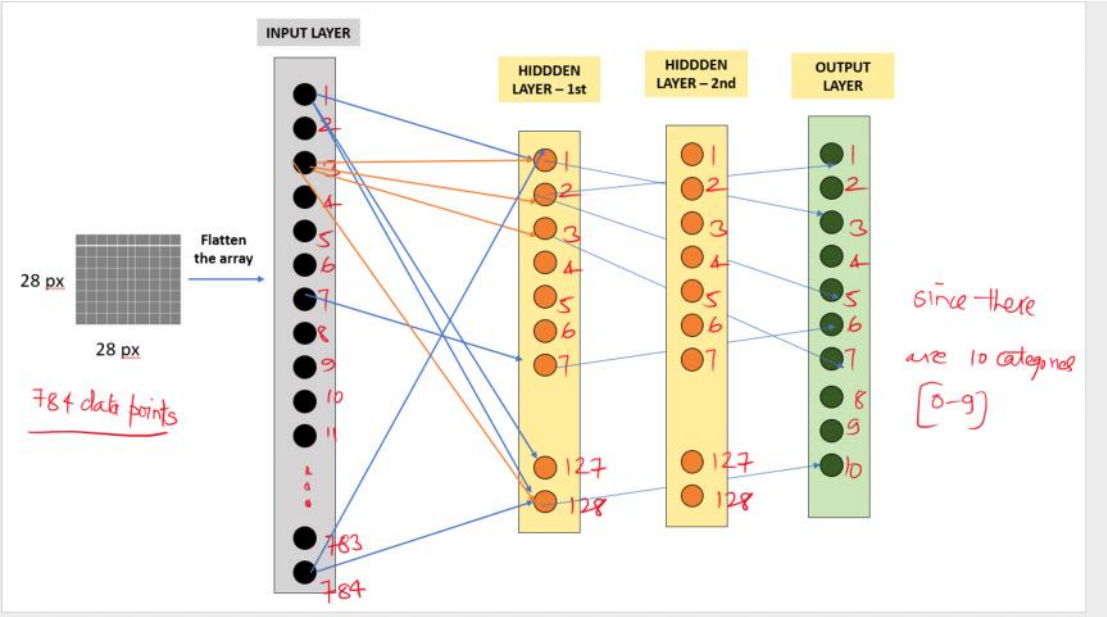
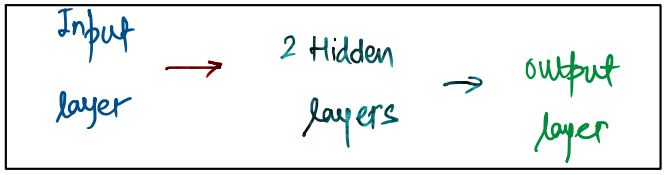
MNIST Code Explanation

01 November 2025 12:24






```
model = tf.keras.models.Sequential() # creates the model object of the class `Sequential` which is basically a linear stack of layers
```

```
model
```

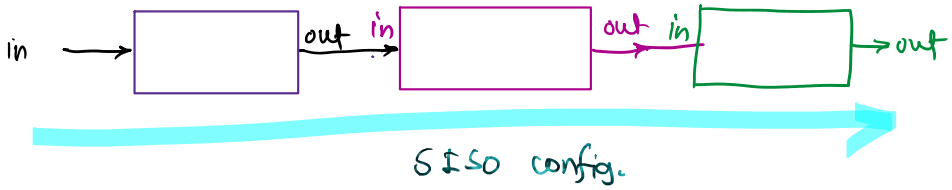
creates an empty sequential model object in TF/Keras

is a linear stack of layers

to add one layer at a time using .add()

Each layer has exactly one input and one output Tensor

SISO: single input single output




```
### Input Layer
```

Raw data for MNIST

```
### Input Layer
model.add(tf.keras.layers.Flatten(input_shape = x_train_normalized.shape[1:]))

### First Hidden Layer having size (no. of neurons = 28)
model.add(tf.keras.layers.Dense(
    units = 128,
    activation=tf.keras.activations.relu
))

### Second Hidden Layer having size 128
model.add(tf.keras.layers.Dense(
    units = 128,
    activation=tf.keras.activations.relu
))


### Output Layer
model.add(tf.keras.layers.Dense(
    units=10,
    activation = tf.keras.activations.softmax
))
```

Raw Data for MNIST

Flattening : Input

Dense layer #1 — Dense(1) : HL #1

Dense layer #2 — Dense(2) : HL #2

output layer — Dense(3): output layer

Sequential Stack.

First Hidden Layer

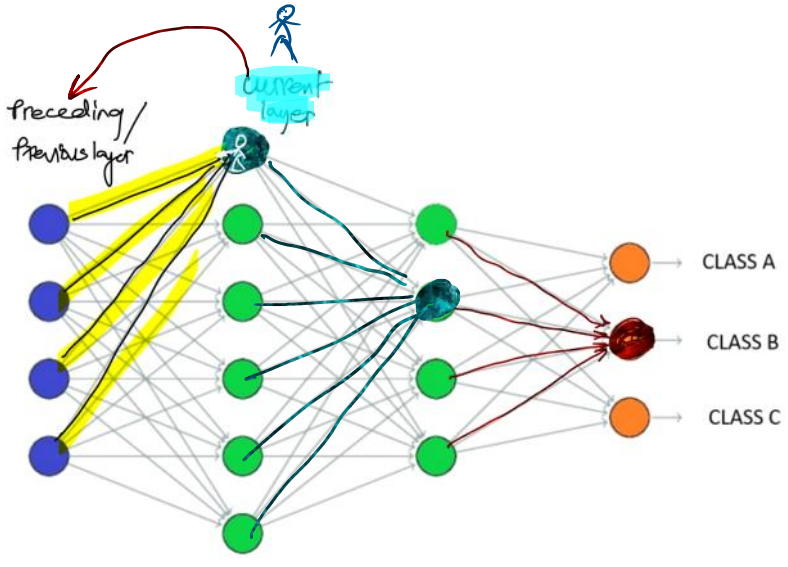
```
### First Hidden Layer having size (no. of neurons = 28)
model.add(tf.keras.layers.Dense(
    units = 128,
    activation=tf.keras.activations.relu
))
```

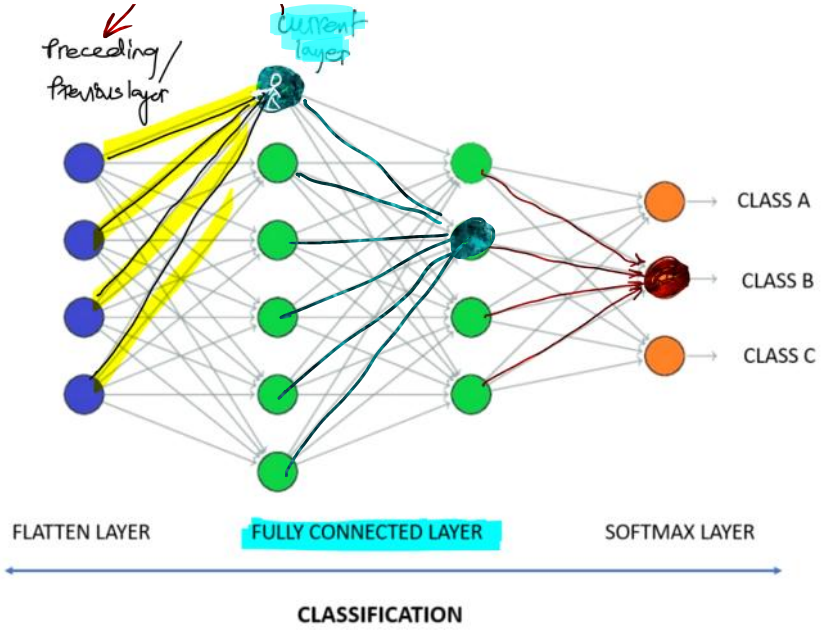
Dense layer #1

Pro tip What is dense layer in a neural network model?

A dense layer is also known as fully connected layer is a fundamental building block in ANN.

In this layer, each neuron in the current layer is connected to every neuron in the preceding / previous layer





https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    lora_rank=None,
    **kwargs
)
```

no. of neurons

RELU, softmax, Sigmoid etc.

by default → TRUE

a technique to initialize weights.

bias initialized with 0 values.

adds a penalty to the layer weights during training (L1/L2) and helps prevent overfitting.

Task: Read about 'glorot_uniform'

Args	
`units`	Positive integer, dimensionality of the output space.
`activation`	Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
`use_bias`	Boolean, whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix.
`bias_initializer`	Initializer for the bias vector.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`activity_regularizer`	Regularizer function applied to the output of the layer (its "activation").
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.
`lora_rank`	Optional integer. If set, the layer's forward pass will implement LoRA (Low-Rank Adaptation) with the provided rank. LoRA sets the layer's kernel to non-trainable and replaces it with a delta over the original kernel, obtained via multiplying two lower-rank trainable matrices. This can be useful to reduce the computation cost of fine-tuning large dense layers. You can also enable LoRA on an existing `Dense` layer by calling `layer.enable_lora(rank)`.

`model.summary()` → To generate a tabular structure of NN model's architecture along with output and # parameters.

Model: "sequential_3"

Layer (type)	Output Shape	Param #
flatten_6 (Flatten)	(None, 784)	0
dense_9 (Dense)	(None, 128)	100,480
dense_10 (Dense)	(None, 128)	16,512
dense_11 (Dense)	(None, 10)	1,290

Total params: 118,282 (462.04 KB)
Trainable params: 118,282 (462.04 KB)
Non-trainable params: 0 (0.00 B)

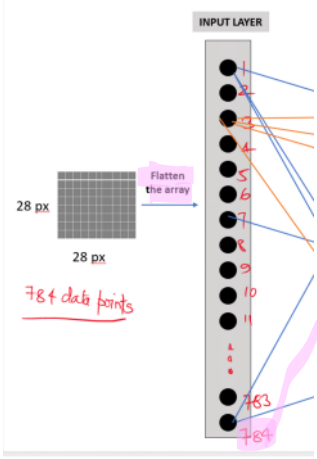
28*28=784

→ no learnable parameters (No weights or bias) in IL

batch_size can vary

↓

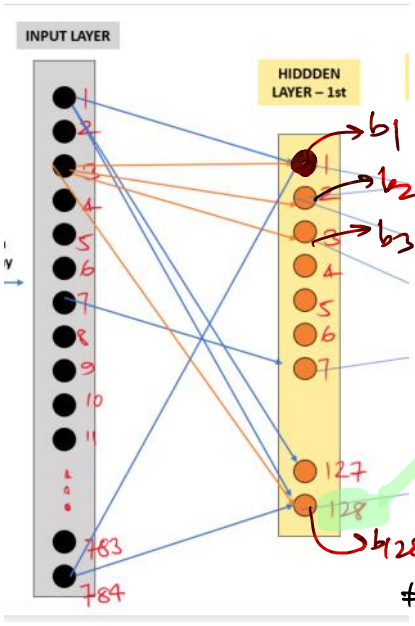
model doesn't know how many samples it will get at a time during training



First Hidden Layer HL#1

First Hidden Layer HL#1

dense_9 (Dense)	(None, 128)	100,480



128 neurons in the HL#1

Input size = 784

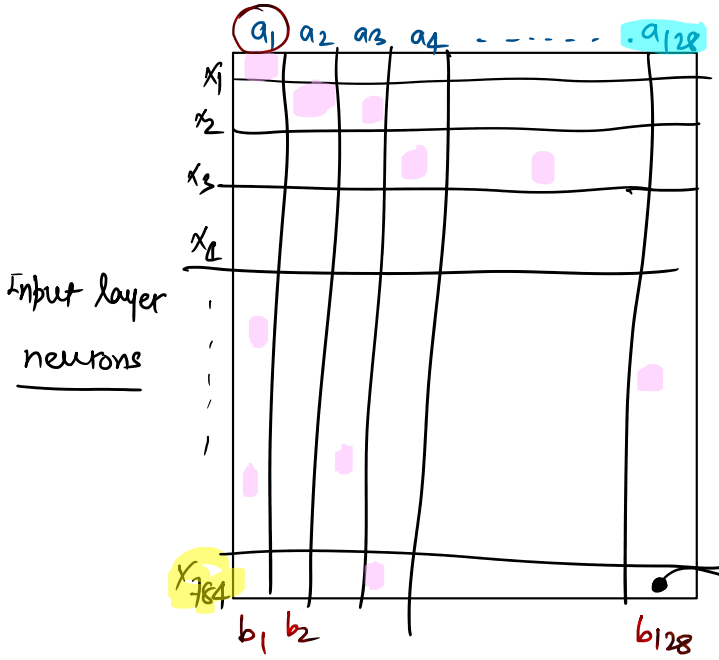
HL#1 size = 128

Weights = 784 * 128 = 100,352

784*128=1,00,352

Biases = 128 ⟶ 128

Total parameters b/w IL & HL = 100,480



⇒ [W . . .]

784 × 128

↓

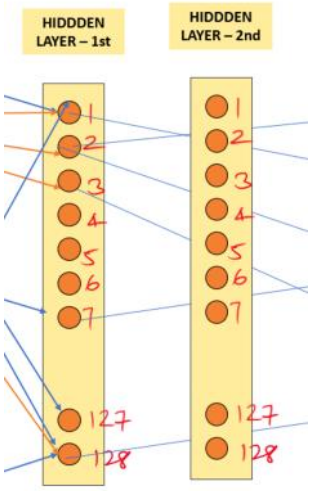
784 × 128 = 100 352 weights)

+ 128

100,480

2nd Hidden Layer

dense_10 (Dense)	(None, 128)	16,512



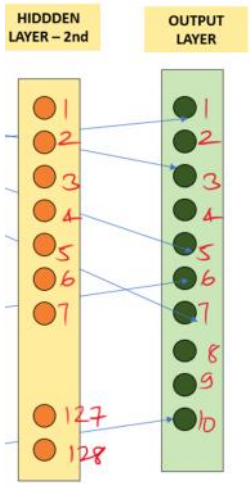
weights bias

$128 * 128 + 128$

$=$ 128 * 128 + 128 = 16,512

output layer

dense_11 (Dense)	(None, 10)	1,290



$128 * 10 + 10 =$ 1280 + 10 = 1290

```
Total params: 118,282 (462.04 KB)
Trainable params: 118,282 (462.04 KB)
Non-trainable params: 0 (0.00 B)
```

→ approximate memory size required to store all model parameters (weights + biases)

118282 parameters

↓

1 byte = 8 bits

dtype : float32 by default

↓

1 param → 32 bits → 4 bytes

dtype : float32 by default

1 param → 32 bits → 4 bytes

118 282 * 4 bytes 118282*4 = 473128 bytes

$\frac{473128}{1024}$ 473128/1024=462.0391 KB

Note: 462.04 KB is NOT the full model size on the disk.
rather it's the size of model's weights only

Compile the model

will discuss later

```
]: adam_optimizer =tf.keras.optimizers.Adam(learning_rate=0.01)
```

a popular choice alpha set to 0.01 for an optimizer

```
]: model.compile(
    optimizer = adam_optimizer,
    loss = tf.keras.losses.sparse_categorical_crossentropy,
    metrics = ['accuracy']
)
```

model.compile() → model is being configured for the training by invoking

- appropriate optimizer
- loss function for multi-class
- metrics set to 'accuracy'
  - given that MNIST is a balanced dataset.

both training and validation simultaneously per epoch.

```
### Train and validate the model
training_hist = model.fit(x_train_normalized, y_train, epochs = 30, validation_data=(x_test_normalized, y_test))
```

[x]test * 60000

[ ]

Training

TF keras does both training and validation once per epoch.

Testing

```
Signature:
model.fit(
    x=None,
    y=None,
    batch_size=None,
```

```
model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose='auto',
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
)
```

→ by default → epoch # 1

per epoch.

each epoch represents one full pass through your training data

```
batch_size: Integer or `None`.
    Number of samples per gradient update.
    If unspecified, `batch_size` will default to 32.
    Do not specify the `batch_size` if your input data `x` is a
    `keras.utils.PyDataset`, `tf.data.Dataset`,
    `torch.utils.data.DataLoader` or Python generator function
    since they generate batches.
```

output/epoch total no. of epochs = 30

```
Epoch 1/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8821 - loss: 0.3846 - val_accuracy: 0.9488 - val_loss: 0.1873
```

mini-batches = 60000/32 = **1875**

1/1875
2/1875
⋮
1875/1875

1 epoch → 1875 updates

In 30 epochs → 1875 * 30 = 56250 updates

epoch # (1/30)
↓
MB # 1
MB # 2
⋮
MB # 1875

Epoch # (1/30) → indicates the training process by epoch
- model is currently in epoch # 1 out of total 30 epochs.

1875/1875 : indicates no. of training batches/epoch.

6S 2MS/step : it indicates epoch # 1 took ~ 6 seconds in total and 2ms on an avg. per batch.

Training

```
- accuracy: 0.8821 - loss: 0.3846 -
```

Testing

```
val_accuracy: 0.9488 - val_loss: 0.1873
```

Per epoch #1

Training accuracy = 88.21%

Training loss = 0.3846

Validation accuracy = 94.88%

validation loss = 0.1873

```
### Train and validate the model
training_hist = model.fit(x_train_normalized, y_train, epochs = 30, validation_data=(x_test_normalized, y_test))
```

training → Simultaneously ← Validation

per epoch.

Note: Default behavior of model.fit()

However you can run all the epochs only for training and then run validation only (30) once.

option #1 Train without validation during epochs.

```
model.fit(x_train_normalized, y_train, epochs=30, verbose=1)
```

Run validation block only once at the end

```
val_loss, val_acc = model.evaluate(x_test_normalized, y_test, verbose=1)
```