Regularisation Methods

[Neural Networks]

- Adam
- L1 / L2
- Dropout
- Batch Norm

Regularisation

in LR, we use

$$L: \min_{\vec{w}, \lambda} \frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2 + \lambda (\|w\|_2)$$

$$\rightarrow L: \max_{\vec{w}, \lambda} f(\vec{w}) + \underbrace{\lambda (\|w\|_2)^2}_{\text{Regularisation}}$$

We can do something similar but here W is a matrix.

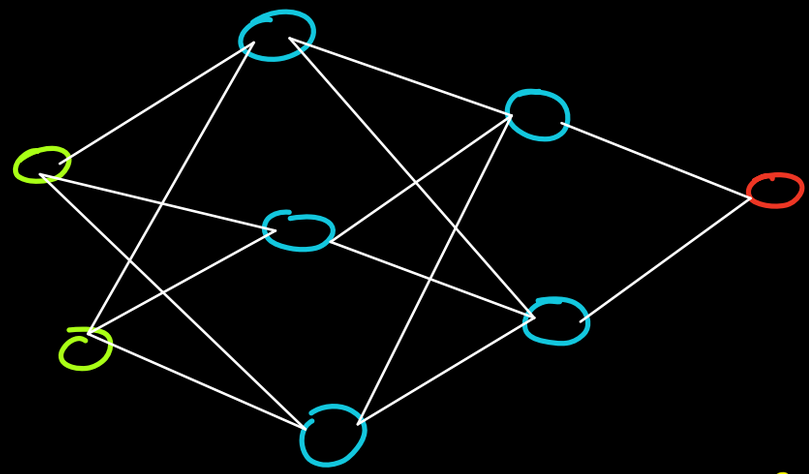
$$\text{Reg}: \underbrace{\lambda \sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij}^2}_{\text{Frobenius Norm}}$$

(for all layers)

(Refer to notes for grad conf)

```
def create_baseline():
    # lambda = 0.01
    L2Reg = tf.keras.regularizers.L2(l2=1e-6)
    model = Sequential([
                    Dense(256, activation="relu", kernel_regularizer = L2Reg ),
                    Dense(128, activation="relu", kernel_regularizer = L2Reg),
                    Dense(64, activation="relu", kernel_regularizer = L2Reg),
                    Dense(1 , activation = 'sigmoid')])
    return model
```

Drop Out



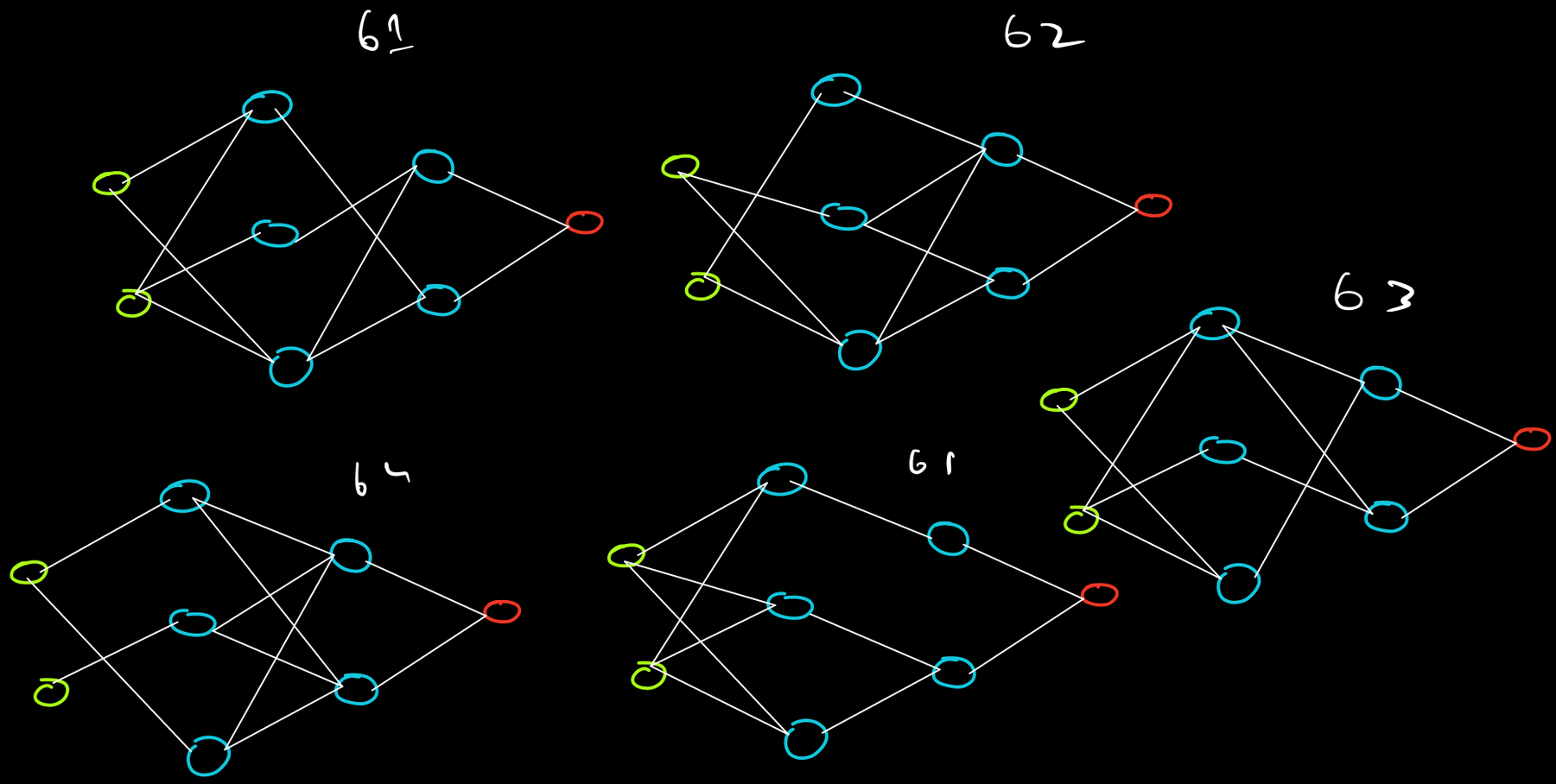
$W_s \rightarrow 2 \times 3$ $\quad 3 \times 2$ $\quad 2$ $= 14$

$b_s \rightarrow +3$ $\quad +2$ $\quad +1$ $= 6$

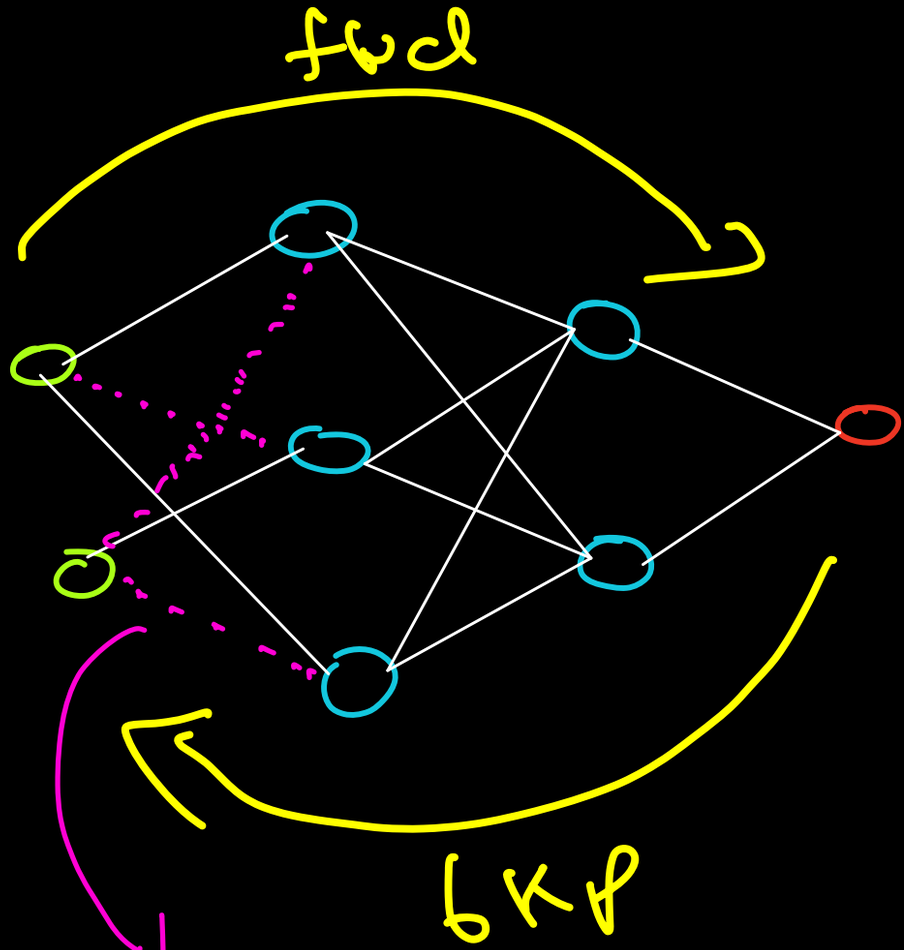
20 params ✓

With deeper NN # params becomes very high. It is common to have more params than training points. This causes overfitting

Solution



→ Randomly remove (drop) connections during each batch.



missing weights wont be updated in this iteration

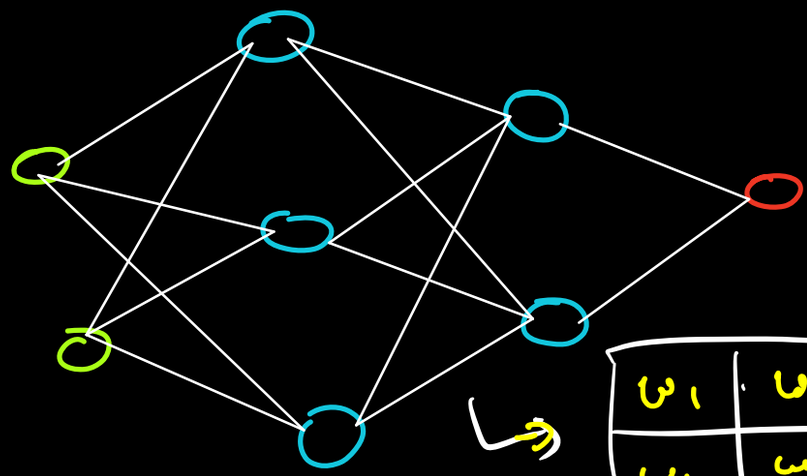
this "kind of" creates an ensemble of many "partially" connected NN.

We know that bagging reduces overfitting (variance)

Each update has less weights now.
So less overfitting.
→ Plus the model needs to perform well for diff combinat^{n} of weights

How is this implemented?

HP → specify dropout rate, eg: 0.2 ↓ 20%



Element-wise mult with random mask

↳

w_1	w_2	w_3
w_4	w_5	w_6

×

1	0	1
1	1	0

Dropout layer has "no learnable" params

During Inference

Since training is done with only eg: 80% of the weights at any given time,

During prediction we use 100% of the weights, but we multiply each weight by say 0.8

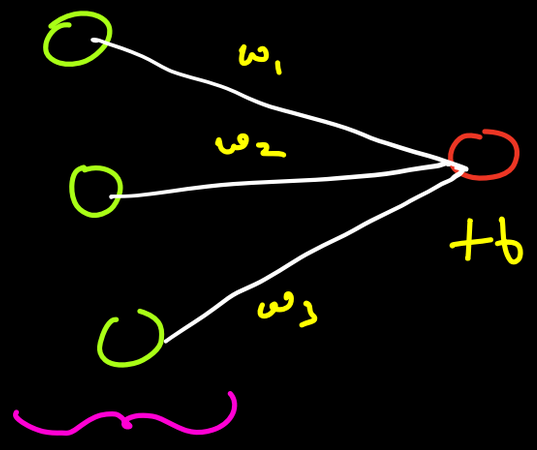
Dropout does not affect biases.

```python
from tensorflow.keras.layers import Dropout
def create_Dropout():
    # lambda = 0.01
    L2Reg = tf.keras.regularizers.L2(l2=1e-6)
    model = Sequential([
                    Dense(256, activation="relu", kernel_regularizer = L2Reg ),
                    Dropout(0.3),
                    Dense(128, activation="relu", kernel_regularizer = L2Reg),
                    Dropout(0.3),
                    Dense(64, activation="relu", kernel_regularizer = L2Reg),
                    Dense(1 , activation = 'sigmoid')])
    return model
```

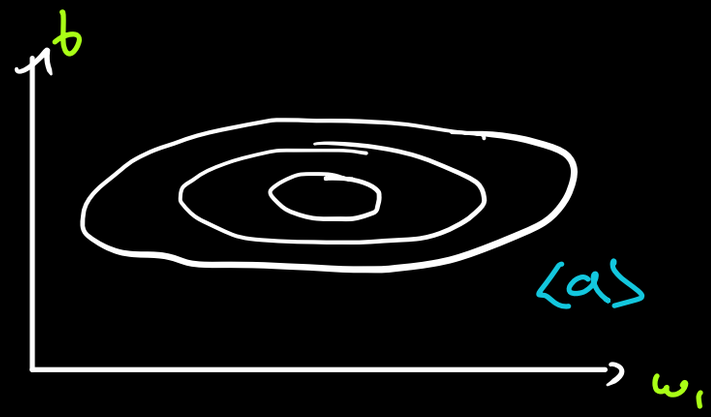
Batch Normalisation

Let's consider a simple NN

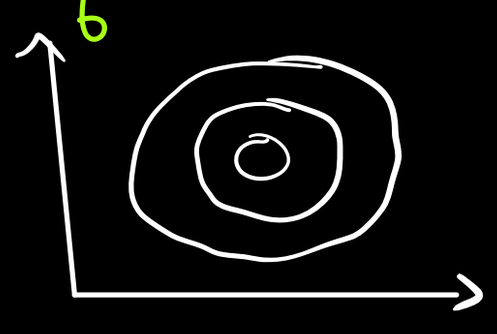
params = 4



Should I scale these values?



Q. Where will it train faster?



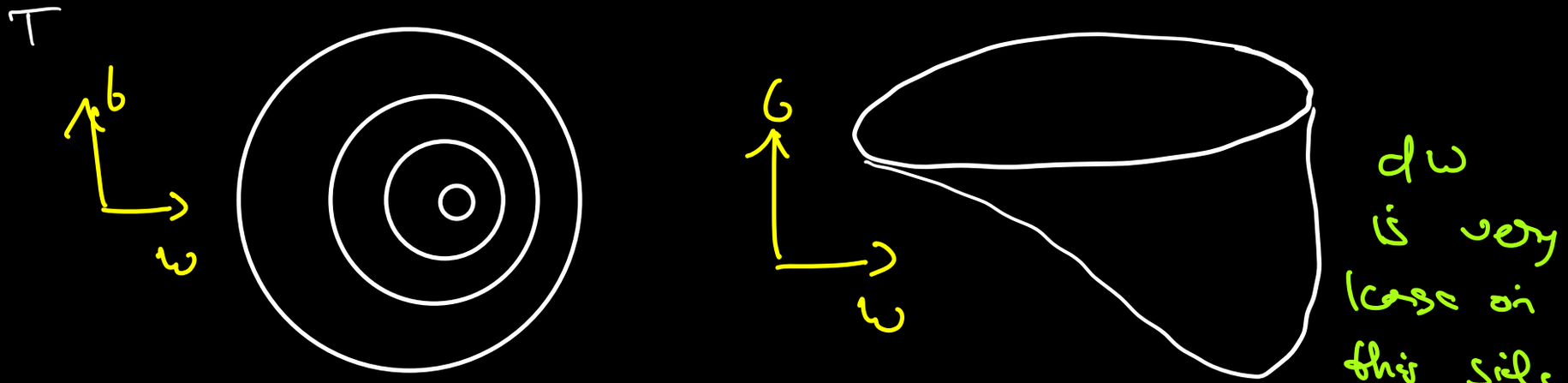
Scaled

→ if we scale the input, that makes w and b ranges similar to each other.

→ It is well known that standardising makes training faster!

But,

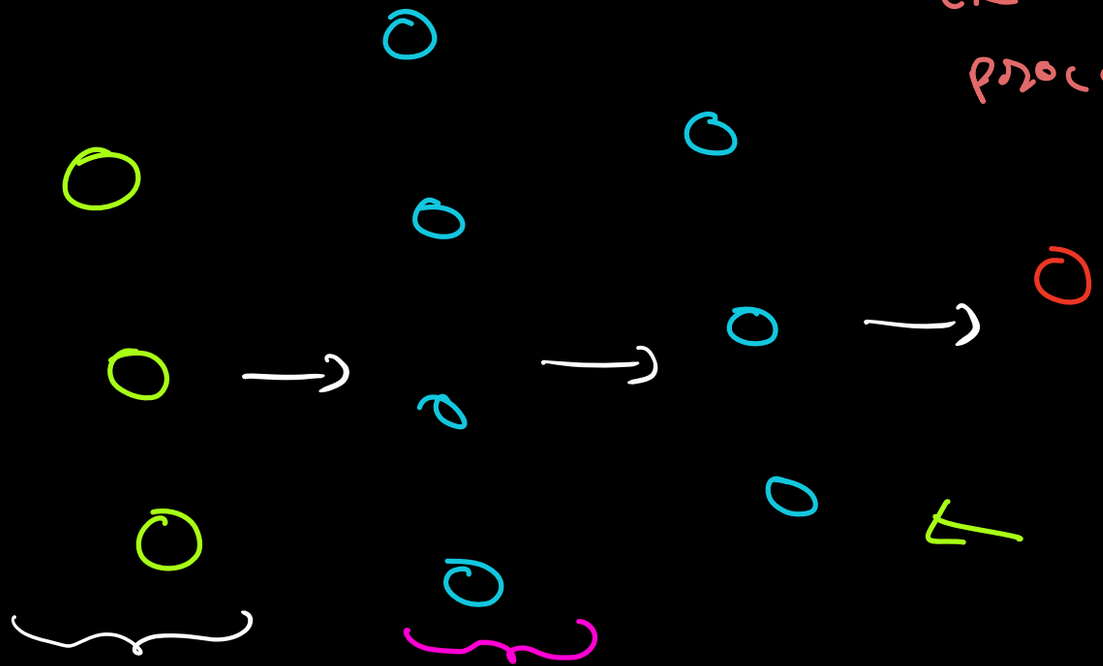
You still need RMS prop / ADAM



So a combinat^n of standardisation and optimisers helps!

Further

Note: Although we are doing standardisat^n the name of the process is BN



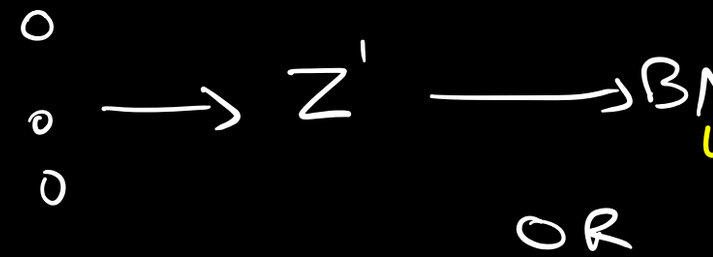
and this too ✓

This is scaled ✓

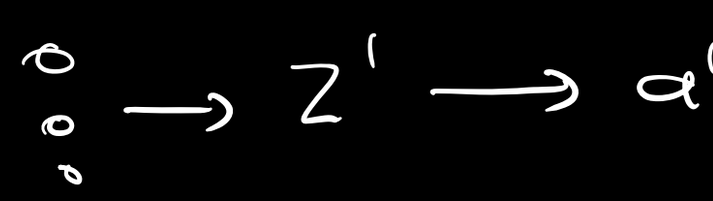
Should I scale this? → Yes ✓

Std scaling the output of a layer is over every batch is called Batch Normalisation

There is still debate on wether to apply it before or after the activation func^n



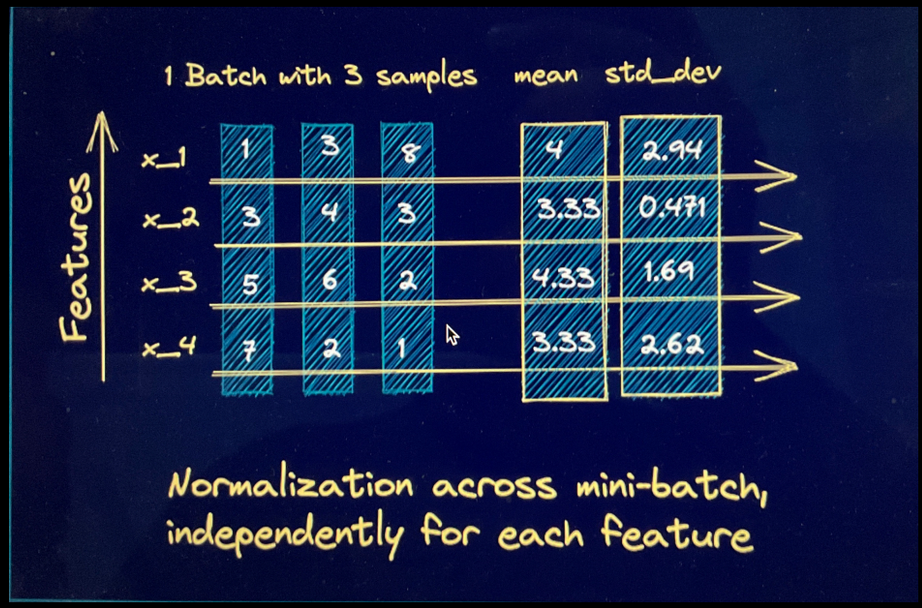
OR



→ You can try this out via code eg:

```python
def create_BatchNormalization_model():
    L2Reg = tf.keras.regularizers.L2(l2=1e-6)
    model = Sequential([
                    Dense(256, kernel_regularizer = L2Reg, activation='relu'),
                    BatchNormalization(),  # <b>
                    Dense(128, kernel_regularizer = L2Reg, activation=None),
                    BatchNormalization(),  # <a>
                    Activation(relu),
                    Dense(64,kernel_regularizer = L2Reg ),
                    BatchNormalization(),
                    Activation(relu),
                    Dense(1)])
    return model
```

How does this work?



We scale one batch at a time! Which means diff batches may have diff μ, σ.

$$\mu_b = \frac{1}{B}\sum_{i=1}^{B} x_i \quad ; B = \text{batch size}$$

$$\sigma_b^2 = \frac{1}{B}\sum_{i=1}^{B} (x_i - \mu_b)^2$$

$$\hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \varepsilon}}$$ → to protect from div by 0

$$y_i = BN(x_i) = \gamma \cdot \hat{x}_i + \beta$$

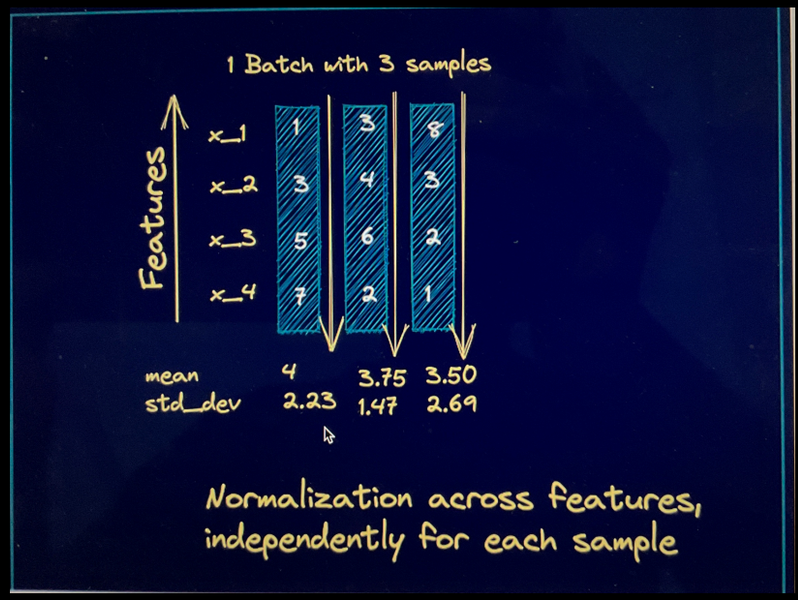
↑ additional params !

If each layer has the output with 0 mean and 1 σ then, we may

not learn much!

So we allow the model to figure out "automatically" the desired mean (β) and std (γ)

Please note that there is also something called Layer Normalisation



But how does this help in regularization?

→ Each batch has a different μ_B & σ_B. This forces the models to learn such weights that perform well on each batch (diff μ, σ) equally (on avg), hence forcing it to generalize better.

Finally,

What μ, σ to use during inference??

→ We take an exponentially weighted avg of all the μ, σ learn over all batches over all epochs.