Class #2:

✒️ Neural Network - Computational model that is inspired by human intelligence.

✒️ Hyperparameter - A hyperparameter is a parameter set before training a machine learning model, rather than learned from the data. It controls the training process and affects model performance.

- Manually set.
- Not learnt from data.
- Settings of the model.

✒️ Hyperparameters & Their Importance

Learning Rate(α)	Controls how much the model updates weights in each step.	0.01, 0.001, 0.0001	Too high -> Model overshoots. Too low -> Model learns too slowly.
Batch Size	Number of training samples processed before updating weights.	16, 32, 64, 128	Small batch -> Better generalization, more noise. Large batch -> Faster but may overfit.
Number of Epochs	Number of times the model sees the entire dataset.	10, 50, 100	Too many -> Overfitting Too few -> Underfitting
Number of Hidden Layers	Determines the depth of the neural network.	1, 2, 3, 10	More layers -> Better feature extraction but more risk of overfitting.
Number of Neurons per Layer	Defines the complexity of each layer.	32, 64, 128, 512	More neurons -> More complexity but higher computation.
Dropout Rate	Percentage of neurons randomly dropped during training to prevent overfitting.	0.1, 0.2, 0.5	Higher values prevent overfitting but may slow learning.
Optimizer	Algorithm that adjusts weights to minimize loss.	SGD, Adam, RMSprop	Different optimizers converge at different speeds and stabilize.

Activation Function	Defines how neurons activate and pass values forward.	ReLu, Sigmoid, Tanh	Affects how well a model captures non-linear relationships.
Weight Initialization	Sets initial weight values before training.	Xavier, He, Random	Poor initialization leads to slow or stuck learning.

✒️ Question - How do we get the number of epochs, number of hidden layers, number of neurons per layer, initialise weights or initialise bias?
=> All of these are derived from trial & error.

✒️ Loss/Cost/Error Function - A function to predict how far our predicted value is from the true value.

$\bar{y} = \sigma z$
$\sigma z = 1 / (1+e^{-z})$
$z = wx+b$
$c = (y - \bar{y})^2$
$\delta c/\delta w = \delta c/\delta \bar{y} * \delta \bar{y}/\delta z * \delta z/\delta w = -2(y - \bar{y}) * \sigma z * (1-\sigma z) * x$
$\delta c/\delta b = \delta c/\delta \bar{y} * \delta \bar{y}/\delta z * \delta z/\delta b = -2(y - \bar{y}) * \sigma z * (1-\sigma z)$

✒️ Loss functions

Mean Squared Error (MSE)	tf.keras.losses.MeanSquaredError	$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$	Regression problems where errors need to be minimized.
Mean Absolute Error (MAE)	tf.keras.losses.MeanAbsoluteError	$MAE = \frac{1}{N}\sum_{i=1}^{N}\|y_i - \hat{y}_i\|$	y - \hat{y}
Mean Squared Logarithmic Error (MSLE)	tf.keras.losses.MeanSquaredLogarithmicError	$MSLE = \frac{1}{N}\sum_{i=1}^{N}(\log(1+y_i) - \log(1+\hat{y}_i))^2$	Regression tasks where small differences matter more than large ones.

Huber Loss	tf.keras.losses.Huber	$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } \lvert a\rvert \le \delta \\ \delta(\lvert a\rvert - \frac{1}{2}\delta) & \text{for } \lvert a\rvert > \delta \end{cases}$	y - \hat(y)
Binary Cross-Entropy	tf.keras.losses.BinaryCrossentropy	$BCE = -\frac{1}{N}\sum_{i=1}^{N}[y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$	Binary classification tasks (e.g. spam detection, medical diagnosis).
Categorical Cross-Entropy	tf.keras.losses.CategoricalCrossentropy	$CCE = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$	Multi-class classification when labels are one-hot encoded.
Sparse Categorical Cross-Entropy	tf.keras.losses.SparseCategoricalCrossentropy	Same as Categorical Crossentropy but with integer labels.	Multi-class classification when labels are integer-encoded instead of one-hot.
Kullback-Leibler Divergence (KL Divergence)	tf.keras.losses.KLDivergence	$D_{KL}(P\Vert Q) = \sum_i P(i)\log\frac{P(i)}{Q(i)}$	Probability distributions in variational autoencoders and reinforcement learning.
Cosine Similarity Loss	tf.keras.losses.CosineSimilarity	$\text{Loss} = 1 - \frac{\sum_{i=1}^{N} y_i\hat{y}_i}{\Vert y\Vert\,\Vert\hat{y}\Vert}$	
Hinge Loss	tf.keras.losses.Hinge	$L = \sum_{i=1}^{N}\max(0, 1 - y_i\hat{y}_i)$	Used for Support Vector Machines (SVMs) and max-margin classification tasks.
Squared Hinge Loss	tf.keras.losses.SquaredHinge	$L = \sum_{i=1}^{N}(\max(0, 1 - y_i\hat{y}_i))^2$	Similar to hinge loss but penalizes large margin violations more.
Poisson Loss	tf.keras.losses.Poisson	$L = \sum_{i=1}^{N}(\hat{y}_i - y_i\log\hat{y}_i)$	Used when modelling count-based data

			(e.g. predicting the number of events occurring).
Log Cosh Loss	tf.keras.losses.LogCosh	$L = \sum_{i=1}^{N} \log(\cosh(\hat{y}_i - y_i))$	Regression tasks, similar to Huber less but smoother.

📌 Regression

- MSE - Mean Squared Error
- MAE - Mean Absolute Error
- Huber Loss

✒️ MSE - Mean Squared Error (MSE) is a commonly used loss function for regression models. It measures the average squared difference between actual and predicted values.

✒️ MAE - Mean Absolute Error (MAE) is a commonly used loss function for regression models. It measures the average absolute difference between actual and predicted values.

✒️ Huber Loss - Huber Loss is a robust loss function that effectively handles outliers by combining Mean Squared Error (MSE) and Mean Absolute Error (MAE).

📌 Classification

- Binary cross-entropy
- Categorical cross-entropy
- Sparse Categorical Entropy

✒️ Binary cross-entropy - Binary Cross-Entropy (BCE) is a loss function commonly used in binary classification problems in machine learning and deep learning. It measures the difference between the true labels and the predicted probabilities.

✒️ Categorical cross-entropy - Categorical Cross-Entropy (CCE) is a commonly used loss function in machine learning and deep learning for multi-class classification problems where each input belongs to one of several categories.

✒️ Sparse categorical cross-entropy - Sparse Categorical Cross-Entropy (SCCE) is a loss function used for multi-class classification when labels are integers. It is an optimized version of Categorical Cross-Entropy (CCE) for sparse labels.

📌 Training Steps

- Forward pass
- Gradient computation / Backward propagation
- Optimization / Update weights & biases

✒️ Forward pass - The forward pass is the process in a neural network where input data flows through the layers, transforming weights, biases, and activation functions, to generate an output (prediction).

✒️ Backward propagation - Backpropagation is an optimization algorithm used to train neural networks by adjusting weights based on the error from the forward pass. It works by propagating the error backwards through the network and updating weights using gradient descent.

✒️ Optimization algorithm - An optimization algorithm in AI is used to adjust the model's parameters (weights and biases) to minimize the loss function and improve performance. A very common algorithm is "Adam".

✒️ Optimizers

Gradient Descent (GD)	$W = W - \alpha \nabla L(W)$	Large datasets with offline batch updates.	Converges to optimal solution.	Slow, requires entire dataset for each update.
Stochastic Gradient Descent (SGD)	$W = W - \alpha \nabla L(W_i)$	Large datasets, real-time updates.	Fast, updates weights per sample.	Noisy updates may not converge.
Mini-Batch Gradient Descent	$W = W - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla L(W_i)$	Balance between GD & SGD	Faster than GD, more stable than SGD.	Still has some variance.

Momentum	$v_t = \beta v_{t-1} + (1 - \beta)\nabla L(W) \quad W = W - \alpha v_t$	Training deep networds with oscillations	Fater convergence, smooths updates.	Requires tuning β.
Nesterov Accelerated Giant (NAG)	$v_t = \beta v_{t-1} + \nabla L(W - \alpha v_{t-1}) \quad W = W - \alpha v_t$	Helps in cases of slow convergence.	Better than Momentum in convex loss surfaces.	Requires extra gradient computation.
Adagrad	$W = W - \frac{\alpha}{\sqrt{G_t + \epsilon}} \nabla L(W)$	Sparse data (NLP, embeddings)	Adapts learning rate per parameter.	Learning rate decreases too much over time.
RMSprop	$G_t = \beta G_{t-1} + (1 - \beta)\nabla L(W)^2 \quad W = W - \frac{\alpha}{\sqrt{G_t + \epsilon}} \nabla L(W)$	RNNs, NLP, and non-stationary loss functions.	Reduces learning rate issues of Adagrad.	Requires tuning β.
Adam (Adaptive Moment Estimation)	Combines Momentum + RMSprop: $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla L(W) \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla L(W))^2 \quad W = W - \frac{\alpha m_t}{\sqrt{v_t + \epsilon}}$	Default optimizer for deep learning.	Fast, adaptive, and works well for most cases.	Uses more memory.
AdamW (Adam with Weight Decay)	Similar to Adam but includes weight decay.	Regularized deep networks.	Prevents overfitting.	Requires tuning weight decay factor.
AdaDelta	RMSprop variant: $W = W - \frac{\Delta W_t}{\sqrt{G_t + \epsilon}} \nabla L(W)$	Avoids manual learning rate tuning.	No need for a fixed learning rate.	Computationally expensive.
Nadam (Nesterov-Adam)	Adam + Nesterov Momentum	Helps with slow convergence.	Combines benefits of NAG & Adam	May not always be better than Adam.

✒️ Activations - In AI, the activation of a neuron refers to the output value of a neuron after applying an activation function to the weighted sum of its inputs. This determines whether the neuron should "fire" and pass information to the next layer.

✒️ Activation Functions

- Sigmoid
- ReLU (Rectified Linear Unit)
- Leaky ReLU
- Tanh
- Softmax

✒️ Activation functions

Activation Function	Formula	Common Use Cases
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	Binary classifications, output later.
Tanh	$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	Hidden laters, better than Sigmoid for centering data.
ReLU	$\text{ReLU}(x) = \max(0, x)$	Most common in hidden layers of deep networks.
Leaky ReLU	$\text{LeakyReLU}(x) = \max(\alpha x, x)$	Avoids “dying ReLU” problem, better for negative inputs.
Softmax	$\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	Multi-class classification, output layer.
ELU	$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^{x} - 1) & \text{if } x \leq 0 \end{cases}$	Deep networks, faster convergence than ReLU.
Swish	$\text{Swish}(x) = x \cdot \sigma(x) = x \cdot \frac{1}{1+e^{-x}}$	Advanced deep networks, often better than ReLU.

Softplus	$\text{Softplus}(x) = \log(1 + e^x)$	Smooth approximation of ReLU, avoids zero gradients.