- apply softmax activation to outputs; converts to probabilities, predicted class highest probability
- movie genre classifier ex; input is actors, genre, director; output vector of probs for each genre; transform raw scores to probs; example
- softmax function: $p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$; x raw output of network, n number of classes; sum of probs is 1, highest prob $\hat{y} = \text{argmax}(p_i)$; loss function cross-entropy
- allows for optimizing one aspect of performance during training; evaluate another aspect that aligns with BI needs
- binary class problem with imbalanced data; cross-entropy used to train but F1 evaluate; better captures balance between precision and recall; MSE vs MAE since MAE more interpretable
- cross-entropy: $L = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$; F1 score: $F1 = 2 \times \frac{precision \times recall}{precision + recall}$; optimizing L minimizes loss, improving estimates; metric measures model performance, evaluation focuses on model's ability to classify correctly
- depends on problem type and desired properties of solution; ensure coefs optimized to minimize chosen error metric
- house price prediction, MSE loss penalizes large errors more severely; leads to optimal coefs for features like size and bedrooms
- LR model $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$; MSE loss function $L(\beta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$; derivative with respect to β_j set to zero is $\frac{\partial L}{\partial \beta_j} = -\frac{2}{n} \sum_{i=1}^{n} x_{ij}(y_i - \hat{y}_i) = 0$; solving gives coefs that minimize MSE, ensuring best linear fit
- some focus on important features or relationships; struggle due to curse of dimensionality, data is sparse and noisy; better ones have built-in mechanisms to reduce impact of irrelevant features
- email span classification where KNN struggles; all far apart since every word is dimension; SVM finds a hyperplane, separates emails (if linearly separable)
- high dimensions distance grows points and becomes less meaningful; euclidean distance between two points in d dimensions: $d(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$; as d increases distances same almost; harder to distinguish near and far points; SVM hyperplane: $w \cdot x + b = 0$ separates classes
- gradient is small, loss function stops improving significantly, fixed number of iterations reached; choice depends on computational resources, desired accuracy, convergence properties
- bucket water ex; stop when almost full (small gradient); water level does not change significantly (loss function improvement); fixed number of refills (iterations); stop too soon bucket not full, wait too long might overflow
- fixed iterations: $t \geq N$, compute efficiency but stop before conv; change in loss function: $|J(\theta^{(t)}) - J(\theta^{(t-1)})| < \epsilon$, stops when diff in loss steps negligible; small gradient: $||\nabla J(\theta)|| < \epsilon$, stops when gradient magnitude below threshold
- good decision boundary cleanly separates classes, performs well on new data; better ones balance between capturing patterns (training data) and flexibility (unseen examples)

- classifying cat and dog images; boundary tight around cat good for training but fail for test (dog is similar); better is margin around for correct in weird examples
- gap between boundary and closest points; larger more robust; margin: $\frac{2}{||w||}$; maximizes margin minimizes $||w||$ objective of SVM; w weight vector of decision boundary
- increase number of layers, add more nodes per layer, different activation functions; improve model's ability to capture complex patterns; risk of overfitting, computational cost, interpretability
- recognize handwritten digits; more layers capture more than basic shapes; more nodes per layer learn subtle details; different activation functions learn different patterns
- $TotalParameters = (n_{inputs} \times n_{nodes}) + n_{nodes} + (n_{nodes} \times n_{nodes}) + n_{nodes}$; more layers or nodes increases parameter count, model fits more complex functions; overfitting as model capacity outgrows data
- capture non-linear patterns in data that linear models cannot; custom layer for non-linear transformations or transform input data before feeding into network
- predict startup success; features are age of founder and initial funding; seem independent but non-linear transformation reveals details; squared funding captures exponential growth potential, interaction between age and funding reveals success patterns
- x input data, $f(x)$ non-linear transformation; linear model $y = w \cdot x + b$, non-linear model $y = w \cdot f(x) + b$; $f(x)$ polynomial, kernel, log transformation; expand feature space to make linearly inseparable data separable
- introduce non-linearity into network; capture complex, non-linear patterns; without activation functions then = linear regression model
- funcs like creative translator in covo; linear coms = raw inputs, funcs is translation adding context and meaning; different funcs mean other translations; ex2 dog image, pixel, edges and shapes, and complex features like fur patterns
- single layer network output: $z = w_1 x + b_1$; activation function sigmoid: $a = \sigma(z) = \frac{1}{1+e^{-z}}$ now not purely linear combination of inputs; $f(x) = g(w_2 g(w_1 x + b_1) + b_2)$ captures non-linear patterns
- enhance performance so all features contribute equally, special case for algos sensitive to feature scales; not beneficial when original scales carry meaningful info
- example of data with height and weight in kg; stand helps ensure neither dominates due to larger numerical range; SVM or GD-based methods benefited; decision trees not affected by feature scaling
- LR coefs estimated as $\beta = (X^T X)^{-1} X^T y$; if x_1 has larger variance than x_2, condition number of $X^T X$ increases; standardizing using $z_i = \frac{x_i - \mu_i}{\sigma_i}$ improves numerical stability and ensures regularization penalties treat all features equally; unnecessary for decision trees, split based on order not magnitude