Neural Networks

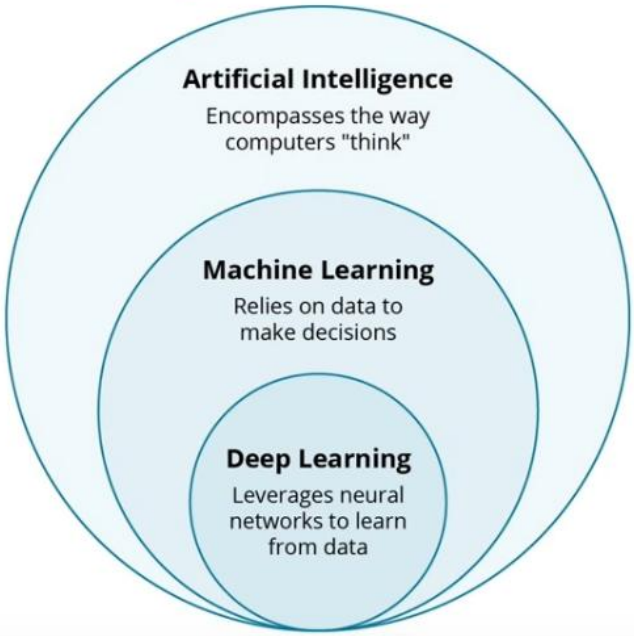
Bhadri Vaidhyanathan – Personal Notes

Contents

Recommended Books – Udacity DeepL Nano: 3
Important Concepts: 3
Underlying Theory and Components; MaxLikelihood, Error Function and Gradient Descent: 4
Maximum Likelihood 4
Cross Entropy (aka Log Loss) 5
Information Theory 7
Gradient Descent: 11
Regularization: 13
How Regularization Helps with Overfitting: 13
Lasso / L1 Regularization 14
Ridge / L2 Regularization 16
Elastic Net Regularization 18
Hyperparameter Tuning: 19
Strategies for Optimizing Hyperparameters 19
Grid search 19
GridSearchCV() 19
RandomizedSearchCV() 19
K Fold Validation: 19
Synthetic Data Generation: 20
1.SMOTE: 20
2. Conditional Tabular Generative Adversarial Networks (CTGAN) 21
3. Data Augmentation via Sampling 22
4. Faker Library for Randomized Data 22
Neural Network 23
Weights and Biases 27
NN patterning Non-Linear relationship: 29
NN Workflow – Single EPOCH: 31
NN BackPropagation 32
Chain Rule in BackPropagation 33
NN and the Architecture FAQ: 34
Data Types and I/O for NN 37
NN – Math/Linear Algebra Layer 37
Activation Functions 38

Step Function: 39
Sigmoid / Logit 41
tanh 43
ReLU 45
Output Layer Function: 45
Softmax 45
Identity Function: 47
Choosing Activation Function: 48
Overfitting and Underfitting: 48
NN Regularization 49
L2(Ridge) Regularization / Weight Decay in NN 50
Hyperparameters: 52
Hyperparameters / Parameters vs Powers of 2? 52
Strategies for Hyperparameter Tuning: 55
Vanishing and Exploding Gradient 56
Weight Initialization 58
Batch Normalization 58
Early Stopping (How many Epochs?): 60
Dropout 60
Random Restart 61
Learning Rate: 61
Optimizing Learning Rate: 62
Momentum 62
Optimizer 63
Re-purposing a NN: 63
Design of Neural Network 64

AI, ML, & Deep Learning



Recommended Books – Udacity DeepL Nano:

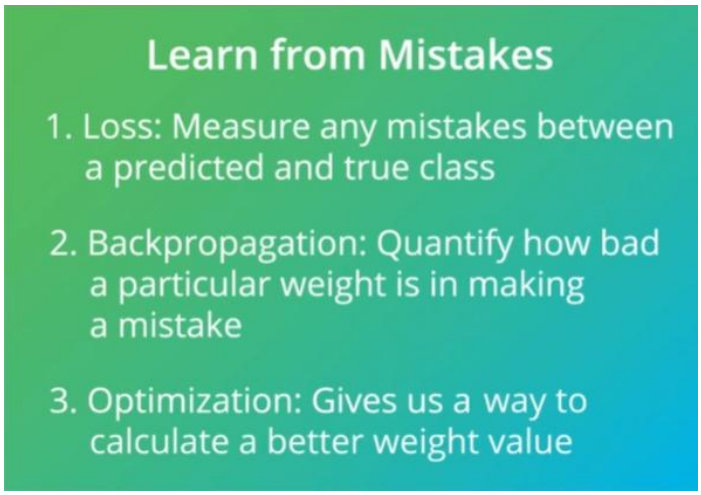
Grokking Deep Learning by Andrew Trask. Use our exclusive discount code traskud17 for 40% off. This provides a very gentle introduction to Deep Learning and covers the intuition more than the theory.
Neural Networks And Deep Learning by Michael Nielsen. This book is more rigorous than Grokking Deep Learning and includes a lot of fun, interactive visualizations to play with.
Inside Deep Learning by Edward Raff is an excellent intermediate book that covers much of the mathematical background, in-depth looks at modern neural network architectures, and develops a good intuition
The Deep Learning Textbook from Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Often simply called "the Deep Learning book", it is a rigorous treatment of the mathematics and theory behind deep learning, in addition to covering a number of important practical issues.
Deep Learning Architectures by Ovidiu Calin is by far the most theoretical book on this list, and the most challenging in terms of mathematical prerequisites. However, if you want to conduct groundbreaking research in neural networks, this book is an excellent reference to have on hand.

Important Concepts:

When to use Neural Networks and when not to:

Picking the Right Tool

When to use Deep Learning	When NOT to use Deep Learning
Images	Simple relationships
Natural language	Tabular data
Lots of data	Relatively little data
Binary classification	
Time series analysis	



Underlying Theory and Components; MaxLikelihood, Error Function and Gradient Descent:

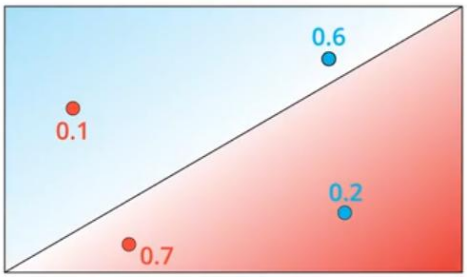
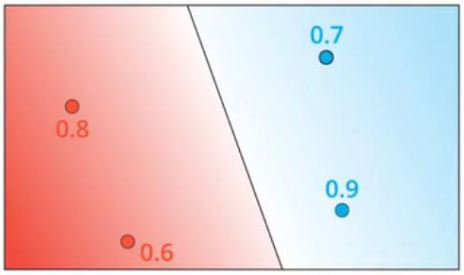
- ✅ **Maximum Likelihood: The theoretical foundation – The probability theory behind fitting data.**
  MLE aims to find model parameters (weights) that maximize the probability of observed data.
- ✅ **Error Function = Cross-Entropy is an Error Function used to estimate loss in Classification cases to achieve MLE**
  Minimizing cross-entropy is equivalent to maximizing likelihood.
- ✅ **Gradient Descent = The Optimization Method to Achieve MLE**
  The numerical method to minimize the loss (maximize likelihood).
  Updates weights iteratively to move toward the maximum likelihood solution.

  Computes gradients of the loss function (cross-entropy) with respect to parameters.

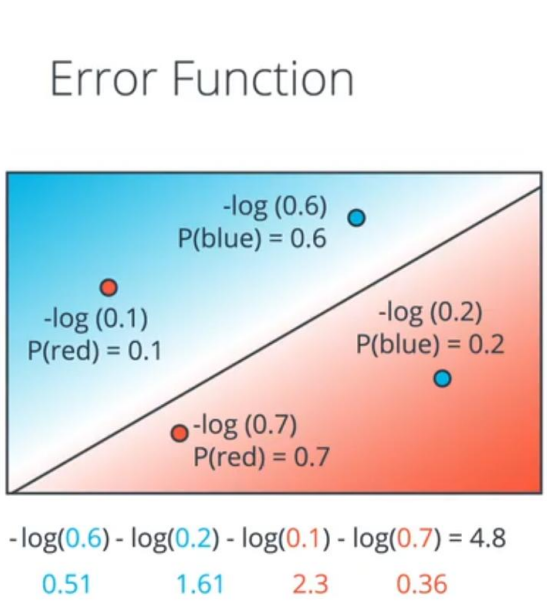
Maximum Likelihood

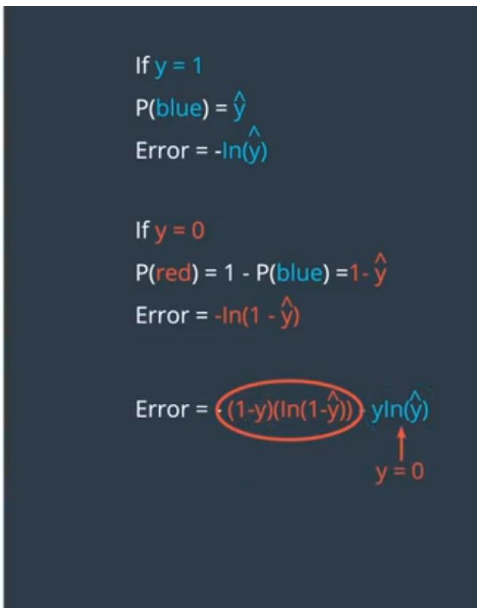
Maximum Likelihood is all about rephrasing the problem statement to machine that the algo's purpose is about a probability (0 to 1) which must be maximised. Probability of each factor or combination of factors (p1 * p2 * p3 etc) and goal is to keep guessing different combinations of probabillities until the highest probability is attained. Rephrasing it into probability is a nice and simple instruction to give to the Machine to help it automatically decide if some thing is better or worser and also the magnitude of how right /wrong it is. This allows the Machine to try different numbers/guess different probabilites, many times over and comparing the guessed/predicted values with the actual to decide which set of guessed values are the best model by simply pick the highest probability number. Maximum Likelihood thus rephrases any real life situation to a binary probability distributions for each input and target variable, trying different numbers and then picking the best numbers with higher probabilities.
Behind the scenes, mathematically , under 1 config, each probability of each input in calculated p1, p2,p3… and then each multiplied like p1 * p2 * p3… to get Total Probability value p. Same is repeated under a different config (coeff values) and Total Prob p is calculated again. Above is simplified explanation of what max. likelihood is about while how it is achieved/calculated is using information theory's Cross Entropy theory – more below.

Config 1: $\sigma (W_1 x + b_1)$ σ is sigmoid function that just scales values to between 0-1 ; a probability value	Config 2: $\sigma (W_2 x + b_2)$
P_x is the probability of a point being the correct color point in correct region (red dot in red region…) Red point in red region will have higher Px than blue dot in blue region which sld explain the different Px below. The line separating the regions into two is a linear line eqn guessed by the algo.	

	
$0.6 * 0.2 * 0.1 * 0.7 = 0.0084$ p_x value of each input is lower if it is classified incorrectly and thus makes the Total Prob value lower	$0.7 * 0.9 * 0.8 * 0.6 = 0.3024$ p_x value of each input is higher if it is classified correctly and thus makes the Total Prob value higher
Config 2 p is higher so it has higher likelihood and algo knows to choose this over config 1. Same is repeated with many configs until the highest Total Prob p is achieved	

Algo is instructed to choose a model which provides the higher prob. The model is trained on train data and all models/configs are attempted and the one picked would be having the max probability.





Source: Udacity Deep Learning Nano course: lesson 11 in Intro section

Cross Entropy is a nice mathematical function that helps defines relationship (or the error which is what we are interested so we can make it an evaluation metric) between Event and Probability of the event. To a Machine, this is an evaluation metric to use and control doing achieves Max Likelihood

Cross Entropy (aka Log Loss)

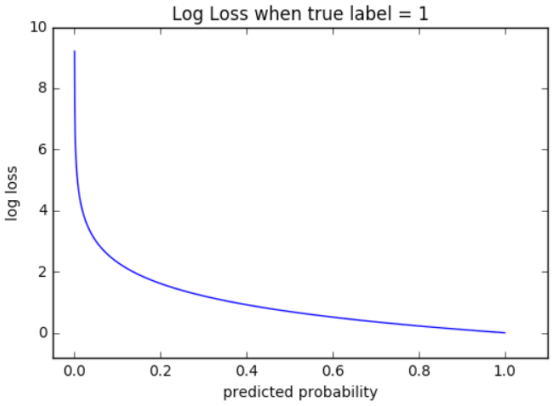
Important Error Function in Classification Neural Networks
Very nice explanation and demo : https://www.youtube.com/watch?v=SKGRzo8VNA4

Cross Entropy is the theoretical framework while Log loss is the mathematical function used hence both terms mean the same in this corner of a domain.

Cross-entropy is a loss function commonly used for **classification tasks** in neural networks. **Cross-entropy quantifies classification error by measuring how far the predicted probabilities are from the true labels.** It does excellent job

to output **a value proportional to the magnitude of the error**. (which is perfect for the design of gradient descent and what GD is expected to do in quantifying error and minimizing)

- Correct → output is 0 or close to 0 value.
- Wrong → output or loss value is high.
- If model is uncertain, then loss value is moderate.



Note: that correct values end up close to predicted prob of 1 whose log-loss values are zero-ish. While, incorrect value that have predicted prob close to zero end up with very high log-loss outputs.

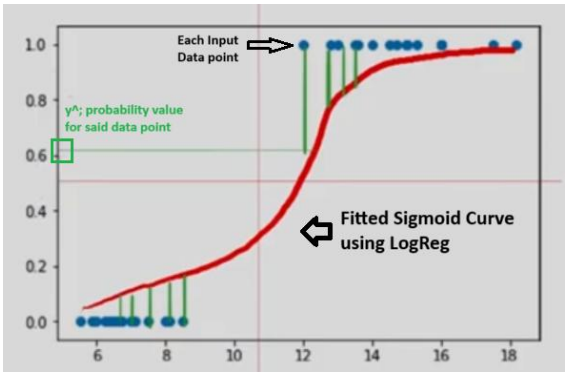
Mathematically, **For a single data point**, cross-entropy loss is:

$$\textbf{CE} = -\sum y * \log(\hat{y})$$

y : the actual class – 1 or 0

y^ : the probability predicted for a point as belonging to 1 or 0 using a sigmoid curve (binary) or softmax(multi-class)

- **Minus is added just cos log values of any number between 0 -1 is negative and to force it to be positive we add a – symbol**



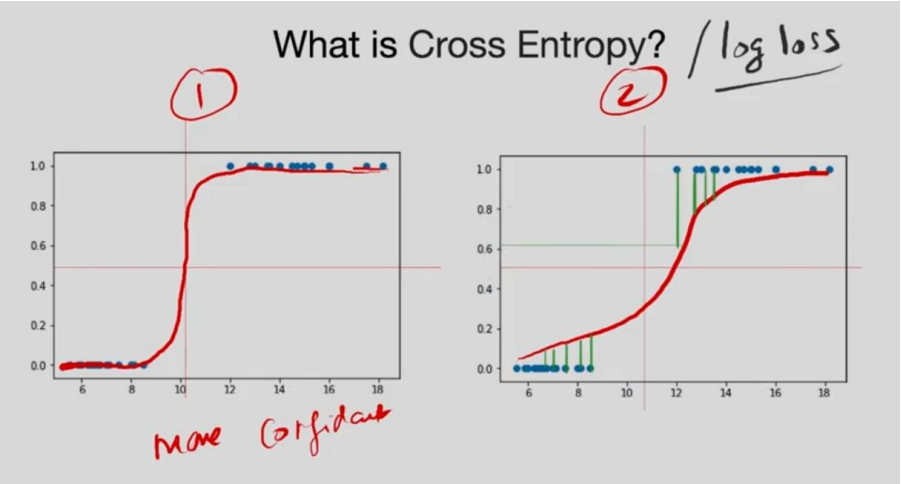
For **binary classification**, the eqn is:

$$CE = -[\, y * \log(\hat{y}) + (1-y) * \log(1-\hat{y}) \,]$$

y and 1-y represent the two classes in question like Blue and Red etc.

Log-Loss or Cross Entropy

Log-loss is an error function used for binary classification that heavily penalizes an incorrect error and gives a minute error to correct ones thus forces a continuous function. Cross Entropy is the theoretical term used in Information Theory and it means the same as Log Loss mathematical function used in Classification NN domain.



Cross entropy in classification is the sum of the log of probabilities (y axis values) of the errors (green vertical lines).

$$-(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

yi is for y=1 cases – green lines in the top right while (1-y) is for the green lines in the bottom left pane.

How Does It Quantify Error?

- **If the model is confident & correct** (y^≈1 for correct class) → **Loss is near 0**
- **If the model is confident but wrong** (y^≈1 for incorrect class) → **Loss is high**
- **If the model is uncertain** (y^≈0.5) → **Loss is moderate**

In SUM, Cross-entropy penalizes incorrect confident predictions heavily, encouraging the model to learn better class distinctions. Which fits perfectly as a gradient

https://machinelearningmastery.com/cross-entropy-for-machine-learning/
https://towardsdatascience.com/cross-entropy-for-dummies-5189303c7735
https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

https://www.rykap.com/math/physics/2017/02/23/entropy-demo/
https://www.youtube.com/watch?v=keDswcqkees

https://towardsdatascience.com/cross-entropy-classification-losses-no-math-few-stories-lots-of-intuition-d56f8c7f06b0
https://gombru.github.io/2018/05/23/cross_entropy_loss/ ; very techy explanation

Prerequisites:

Information Theory

Information Theory by Claude Shannon
Bits 1 and 0 are the basis of all information. It by itself about any event represents the probability of that event. Eg: Event A –

- Models estimate probability distributions of the data
- Parameters, variables, weights are all information in and about the data
- Models capture the information we need from the data in our parametersS

Considering that 1 and 0 represent the probability of an event:
The (Useful) Information provided by the event is

I(X) = - log p Aka probability that event occurred

- **Minus is added just cos log values of any number between 0 -1 is negative and to force it to be positive we add a – symbol**

Cross Entropy:

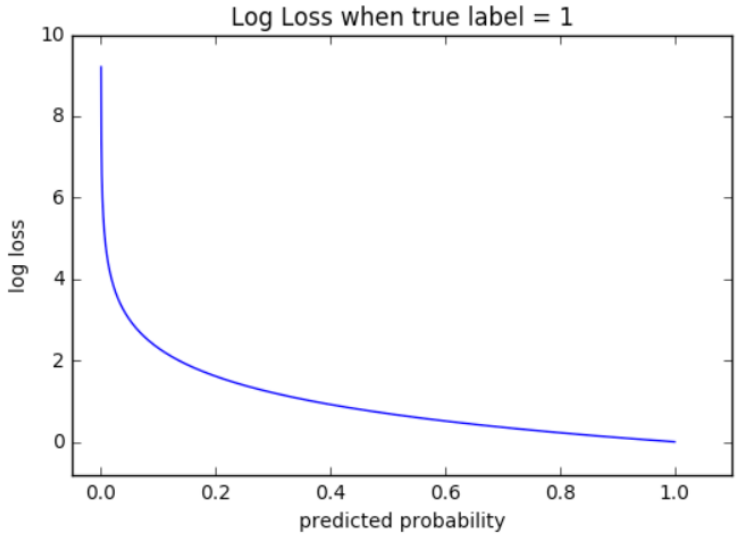
CE measures the **difference** between the predicted probability distribution (y^) and the actual probability which are the class labels (y) themselves 1 or 0.

Cross Entropy = Comparing actual class (1 or 0) y_i vs Predicted Prob. values p_i

Cross Entropy = $$-\sum_{i=1}^{N} y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

- Which is quite similar to Shannon's entropy formula above
- Y is the actual label value and Y is best for simplicity sake be 1 or 0 else above formula wont work
- Y can be multi-class wch makes the math more complex but it is doable
    - https://www.youtube.com/watch?v=keDswcqkees
- Miniziming above formula in effect maximizes the log likelihood of the model itself (again only for binary classification (1 & 0 target variables)

- Cross entropy is comparison between two probability distributions.
- Consider two probability distributions a (ml predicted probability values from softmax func) and b (actual values 1 or 0).
- Entropy is a measurement of uncertainty / a formula to at best quantify uncertain events.
- Cross entropy is a loss function that can be used to quantify the difference between two probability distributions.

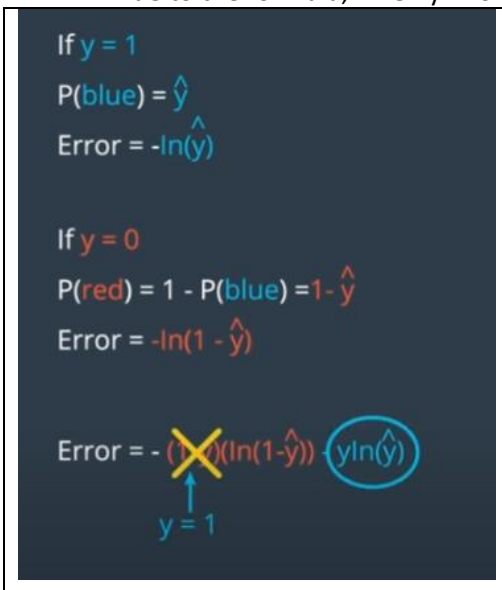


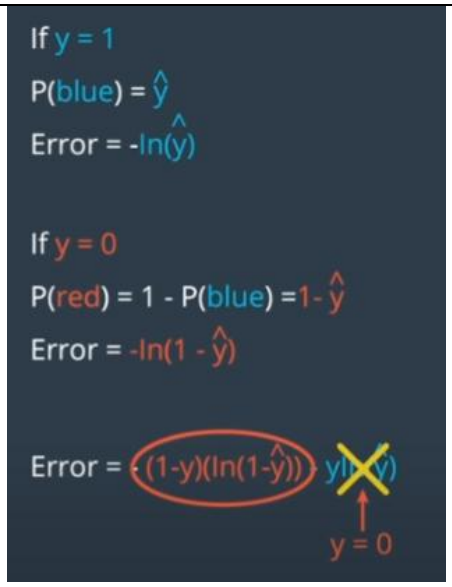
The graph above shows the range of possible loss values. As the predicted probability approaches 1, log loss slowly decreases. As the predicted probability decreases, however, the log loss increases rapidly. Log loss penalizes both types of errors, but especially those predictions that are confident and wrong!
Cross-entropy and log loss are slightly different depending on context, but in machine learning when calculating error rates between 0 and 1 they resolve to the same thing.

Cross entropy usage :

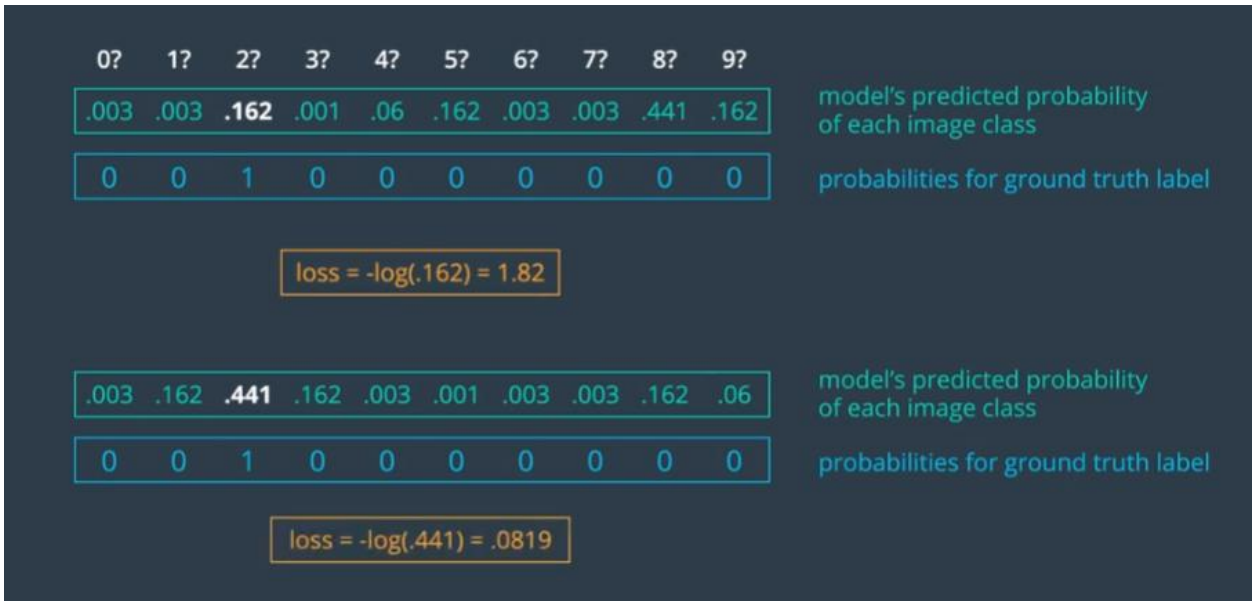
$$\text{Error Function} = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)(\ln(1-\hat{y}_i)) + y_i\ln(\hat{y}_i)$$

- Consider binary classification situation of predicting blue or red
- y refers to actual values (training data) while yhat refers to ML predicted values. Basically two prob. distributions that will be compared via cross entropy
- Above is the cross entropy formula for this case
    - 1-y is the probability of red while y is the probability of blue as per training data (actual values)
    - Ln.(1-yhat) and Ln.yhat are log probs. as per ML predicted values
- The genius bit is that since this is Binary classification where one class is 1 and other is 0
  Due to the formula, when y=1 or y=0, one of the terms cancels to 0





In Multi-Class Classification / Softmax output:



Above are pics of two models with their predicted prob outputs (using softmax in multiclass while it will be sigmoid if binary) that suggests different probs for different classes.

Ex. Top model prediction is very uncertain (many probs close to each other 0.4, 0.1 instead of 0.9, 0.1 etc) and infact wrong since 2 is the answer but it looks like model thinks it is an 8. Thus this leads to a high Cross Entropy / log loss number.
Below model is correct and this leads to a low cross entropy number.
This Cross entropy number is what the model tries to reduce with GD and Back propag just like how Linear Reg (OLS) method) tries to reduce the SSR formula using differentiation analytical technique.

Better than MSE for Classification

- **Mean Squared Error (MSE)** results in slow convergence and poor performance in classification.
- Cross-entropy **avoids saturation issues** that MSE suffers when probabilities are near 0 or 1.

What makes Cross-Entropy a uniquely good error function for classification in Neural Networks?

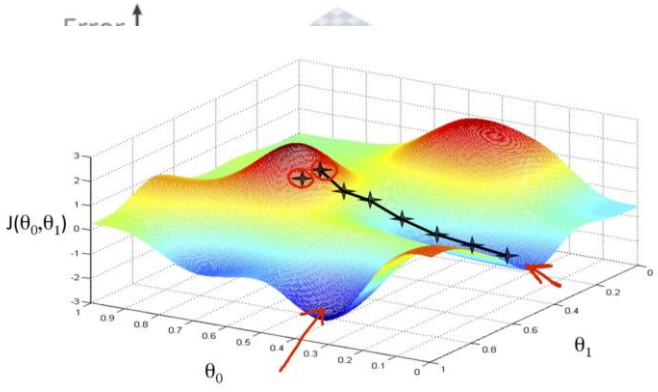
- Minimizing cross entropy is equivalent to finding a model of maximum likelihood

The "Cross" in name comes from comparing two distributions instead of just one (like in standard entropy formula).

<u>Why are Error Continuous instead of Discrete?</u>
Errors could be technically discrete (2,4,6…) as per the case but that creates a lot of ambiguity in GD and so it is forced to be continuous. We need to be able to take very *small* steps in the direction that minimizes the error, which is only possible if our error function is **continuous**. With a discrete error function (such as a simple count of the number of misclassified points), a single small change may not have any detectable effect on the error.

"Error Function" Chart

Charts the error with the diffe derstand which is the best combination of weights to sel



W1, W2, θ_0, θ_1 are weights/coeffs while Y axis(vertical) is the error for each combination. These show the hills and valleys in viz form for easy understanding and help one quickly like elbow chart and thus understand which weights are the best.

Gradient Descent:

The Optimization Method used to achieve Max Likelihood Estimation.

"Gradient" = In Calculus, gradient is the direction of the steepest / highest increase.
gives the value as to how the weights' value should be increased or decreased to get the maximum error.
Gradient is like slope in linear reg. eqn telling the rate of growth at that point.

"Descent" = is the direction and -ve of the Gradient calculated and also implies going down slope to lower error region / movement downwards to a minima

"Gradient Descent" = in short, is the negative of the gradient of the error function at a point. It basically, tells you how the weights should be modified in optimal way to reduce the error. In the end, a single epoch of Gradient Descent, gives you new weights where each update is customized for each weight with ref. to the Error.

In long, Gradient is a vector value that tells which direction (values must change +ve or -ve) and the amplitude/strength/steps (0.5 step or 5 steps) which will increase the error the most. Hence -ve of the Gradient thus GDescent tells optimal way to reduce error. Mathematically, it is the gradient of E (error) given by the vector sum of the partial derivatives of weights with respect to E.

If there are two weights W1 and W2 then G(E) is vector sum of the partial derivatives of the weights (dE/dW1, dE/dW2) and bias B [dE/dB] with respect to E. Bias here is like the intercept or constant like c constant in y = mx+c eqn.

Gradient of Error, G(E) = Vector Value (partial derivative of W1 , partial derivative of W2, partial derivative of bias B...)

= Vector Value ([dE/dW1] , [dE/dW2], [dE/dB]...)

Gradient Descent = -1 * G(E)

Gradient is a Vector:

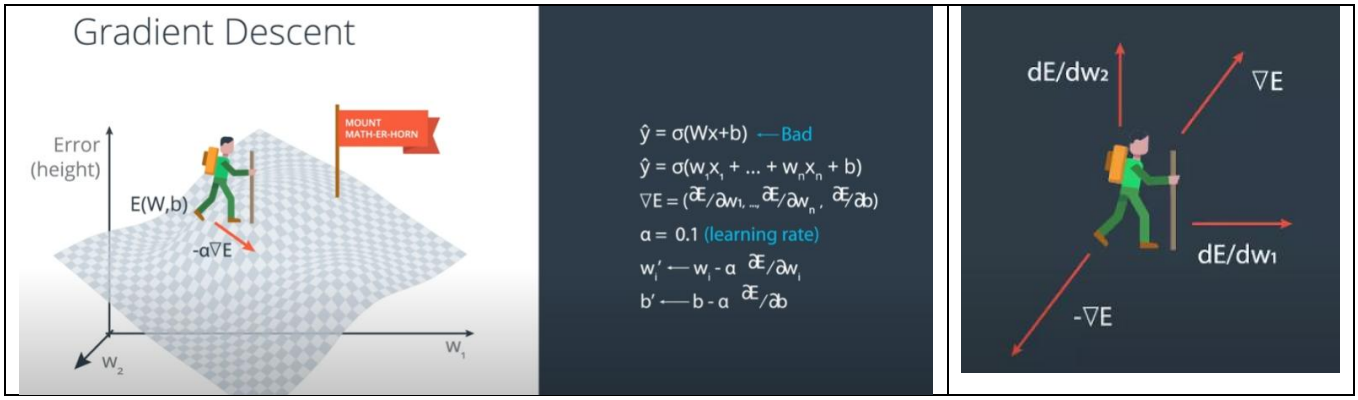
Gradient is a vector value that keeps the partial derivatives of each weight independent thus each weight is updated independently together in one go. **Mathematically efficient in arriving at complex non-linear relationships quickly.** Which is where Neural Networks or GD Rocks!!

When Gradient Descent has to be applied to the weights and bias, since it is in vector form, each individual constituent [dE/dW1] or… or [dE/dB] is independently applied to its corresponding weight or bias to get the new weights and bias with a Learning Rate control element Alpha α.

$$W_{1new} = W_{1old} - \alpha * [dE/dW_{1old}]$$

...

$$B_{new} = B_{old} - \alpha * [dE/dB_{old}]$$



1. To visualize Gradient Descent, first, it is the relation between weights and error. Put it on paper with y axis/ vertical as error while other axes are weights – no independent variable here.
   Independent variable x is coming from training data and will have its own natural distribution. For the given distribution of inputs (independent vars) finding the optimal weights (lowest error) is all this is abt.
   a. w is weight basically "the value" given to each independent variable. this weight is increased if the variable is important to target variable or decreased if not. Tinkering the weights is all GD is abt.
   b. Yhat is the target variable, while w_1x_1, w_2x_2, w_3x_3… are all the weight*independent variable
   c. b is bias like the c in y = mx +c. above w is not m(slope)
2. Gradient = Derivative (if only one variable or weight) else Partial Derivative (multiple var or weight) ; Gradient like slope tells rate of growth (as in defines eqn going up in positive sense) while we need direction to go down so we take negative of gradient
3. Learning rate α is control element we force into the equation that allows to control how much is applied each iteration. It is intentionally makes learning slow which is more optimal in ML cos else so we will jump around overshooting. Without it, high variable values of dE/dW might make giant leaps and make us leap back en forth without converging on optimal results slowly. Empirically found to be more optimal way of arriving at optimal values.

Gradient Descent Algo:

GD = Optimization Algo that uses GD method to gives us a way to improve the estimation of weights in NN.

SGD = optimizer Algo which helps NN judge the results/errors and decide, how each weight is to be updated

Any algo that tries to minimize error and find minima. Stochastic GD is a type of GD as is Momentum, Adagrad etc.

How?
Gradient of the loss with resp. to weights. Tells how bad a weight is and thus points which nodes/weights are the bad ones that led to errors and help us correct them. Good ones also get an update but very small – close to zero change. This is what brings about the non-linearity in the detection. GD then allows us to calculate a better a weight value.

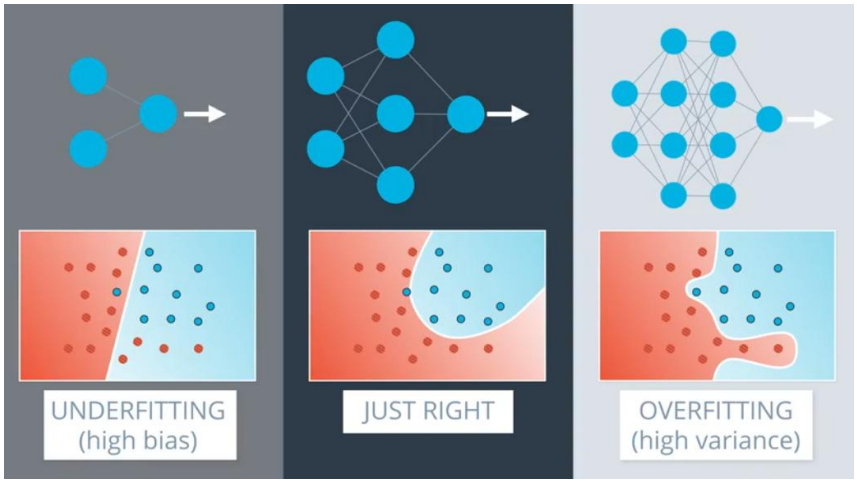
Stochastic Gradient Descent (SGD) and Batch Gradient Descent (BGD) :

Type	Update Frequency	Batch Size	Computation	Convergence	Noise
Batch	Once; After processing all training data	(Full dataset)	Slow (large memory use)	Smooth, stable	Low (since it uses all data)
Stochastic	Every Item; After each training run	1	Fast (updates frequently)	Noisy, fluctuates	High (due to single update per example)

Mini-Batch Gradient Descent (most commonly used in deep learning) → Uses a **subset** of the dataset (e.g., batch size = 32, 64, 128) to balance speed and stability.

Regularization:

Too Many or Too Large Coeffs → Overfitting
Solution: Regularization → add a function of coeffs to Error so if too many or too large coeffs happen, the error is high leading to networks that relegate less useful variables and ensures few of them don't dominate.



https://www.youtube.com/watch?v=VqKq78PVO9g
Regularization is a method to improve **overfitting** and can be used, in limited sense, to reduce underfitting as well. Regularization is technique to induce/increase bias thus making line more straight the same technique could be used to **reduce bias/increase variation to certain limit.** Regularization basically smoothes the curve and so reduces variance and increases bias. To mitigate overfitting, increase regularization and get more smoothing by increasing lambda hyper param.

Regularization, like L1 and L2, is primarily designed to help prevent **overfitting**. By adding penalties to the model's complexity, regularization forces the model to simplify, helping it generalize better and not overfit too much.

How Regularization Helps with Overfitting:

Models at times get over fit such that they look ok with training data but with test or unseen data, they have really bad accuracy. Ex. During training accuracy is 66% but with test data it is 13%. OVERFITTING!!!There are three popular regularization techniques, each of them aiming at decreasing the size of the coefficients.

Lasso and Ridge achieves their task by adding an artificial variable to the cost function with purpose to penalize too many variables or too large coeffs and this controls how an algo improves itself during the numerous iterations.

- **L1 Regularization** (Lasso) encourages the model to reduce some feature weights to zero, effectively removing irrelevant or redundant features, which simplifies the model.
- **L2 Regularization** (Ridge) spreads out the importance across all features by shrinking the dominant feature weights, preventing any single feature from dominating the model.
- Elastic Net, a convex combination of Ridge and Lasso.

Does Regularization Help with Underfitting? Not directly. Underfitting usually happens when a model is too simple to capture the underlying patterns in the data. Regularization, increases bias (more straightness), can sometimes **worsen underfitting** by making the model even simpler.

Lasso vs Ridge:

1. **Lasso tends to do well if there are a small number of significant parameters and the others are close to zero** (ergo: when only a few predictors actually influence the response).
2. Ridge works well if there are many large parameters of about the same value (ergo: when most predictors impact the response).

Lasso / L1 Regularization

An extra term is added to Error/Cost/Loss function. The extra term is a function of coefficients after most recent training run. This extra term which during gradient descent differentiation and update of the coefficient, makes insignificant coefficients to zero thus cancelling a factor/predictor. Effective for in-built feature selection.

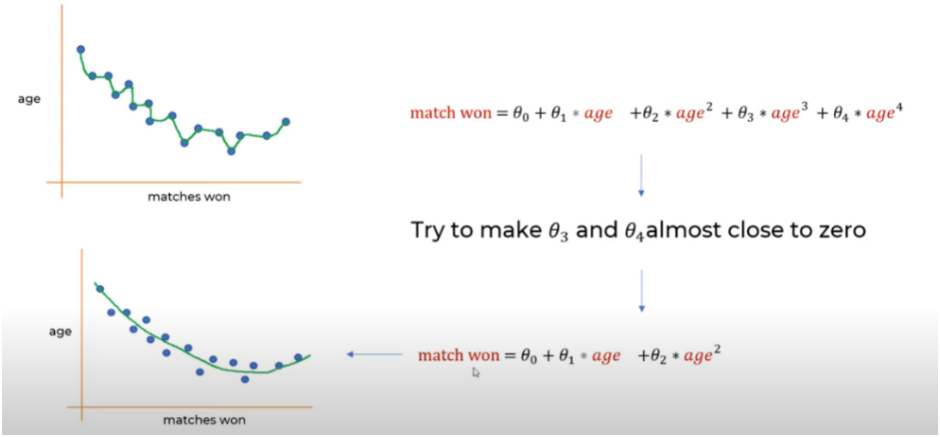
When/How?: Coefficients with small gradients from the loss function (low feature importance) are disproportionately affected during grad.descent coefficient optimization by the L1 penalty and during each of the iterations, leading them to shrink to zero faster.

L1 Regularization encourages sparsity (i.e., some coefficients becoming exactly or near zero) and thus made insignificant.

L1 Regularization

$$mse = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - h_\theta(x_i)\right)^2 + \lambda \sum_{i=1}^{n}|\theta_i|$$

3. If you have 2000 columns and rows, which so much dimensions (thus leading to overfitting), linear model should be good.
   Lasso Linear Reg is best method to shrink the beta/coeff of many unuseful factors to zero
4. By using regularization, the unnecessary coeffs Θ_3 , Θ_4 get reduced to zero or very very small value close to 0

5. 

How L1 Regularization Works

Let us see how L1 Regularization works with an example. Lets take MSE which is a popular loss function used by algos to decide on how to proceed and find the best solution (best line in Linear. Reg.).
Loss/Cost Function = MSE (typical formula which is not important here)

1. **Cost Function with L1 Regularization**:
    - In case of L1 Regularization the Loss/Cost Function gets added λ (lambda) and θ (a func of coefficients)

      Loss/Cost Function = MSE + λ * ∑|θi|
    - λ∑|θi|: Penalizes large coefficients, where lambda λ controls the penalty strength
    - **In L1, |Θ| is modded and only ∑|θi| absolute value is taken**
2. **Optimization with L1 Regularization**:
    - The optimization algorithm (e.g., gradient descent) seeks to minimize the cost function.
    - During optimization, the penalty term **λ∑|θi|** applies extra "pressure" to reduce coefficient magnitudes.
    - L1 regularization encourages sparsity because the gradient of |θi| is **constant** (not dependent on θi) for non-zero values, making it easier for small coefficients to shrink to zero.
3. **Effect on Unnecessary Coefficients**:
    - Coefficients corresponding to less important features (e.g., θ3 and θ4) receive smaller gradients from the loss function.
    - The regularization term dominates for these smaller coefficients, shrinking them faster toward zero.
    - Once a coefficient reaches zero, it stays there because the gradient of |θ| is not defined at zero, effectively stopping further updates.

Numerical Example

Let's assume we have the following setup:

- **Loss/Cost Function = MSE + λ∑|θi|**
- Initial coefficients: θ1=4, θ2=3, θ3=0.5, θ4=0.1 where θ3 and θ4 are for less important features.

Step-by-Step Optimization

1. **Initial Gradient Update**:

- For Loss Function, gradient updates for θ will be proportional to feature importance.
- For $\lambda \sum |\theta_i|$, the gradient for each θ_i is:

$$\frac{\partial}{\partial \theta_i}(\lambda|\theta_i|) = \lambda \cdot \text{sign}(\theta_i)$$

    - For θ>0, this is λ.
    - For θ<0 this is −λ.

2. **Gradients for Each Coefficient**:
    - Assume gradients from the loss function are g1=−2, g2=−1, g3=−0.1, g4=−0.01.
    - Combined gradients: θ1:−2+λ, θ2:−1+λ, θ3:−0.1+λ, θ4:−0.01+λ
3. **Effect of Regularization (λ=0.2)**: **After one update (many happen during optimization)**
    - θ1 4→7.2
    - θ2 3→2.2
    - θ3 0.5→0.4
    - θ4 0.1→0
4. **After Several Iterations**:
    - θ3 and θ4, which have smaller initial gradients and are associated with less important features, shrink faster due to the regularization penalty dominating their updates.
    - Ultimately: θ3→0,θ4→0

Key Takeaway

L1 regularization drives unnecessary coefficients toward zero because:

- The penalty λ|θ| applies consistent "pressure" regardless of the magnitude of θ.
- Coefficients with small gradients are disproportionately affected by the penalty, leading them to shrink to zero faster.

This sparsity property makes L1 regularization effective for feature selection.

Ridge / L2 Regularization

An extra term is added to Error/Cost/Loss function, which during gradient descent differentiation and update of the coefficient, makes dominant coefficients less dominant and in way normalizing all the factors. Effective for in-built normalization of features, in models where all features contribute some predictive value, even if minimally.

When the Θ^2 is squared

L2 Regularization

$$mse = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - h_\theta(x_i)\right)^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

How L2 Regularization Works

1. **Cost Function with L2 Regularization**:
    - **Loss/Cost Function = MSE + λ ∑ θ_i^2**

- $\lambda \sum \theta_i^2$: Penalizes large coefficients more severely because the penalty grows quadratically with the magnitude of θi.

2. **Optimization with L2 Regularization**:
    - The optimization algorithm minimizes the cost function and during this optimization by updating each coefficient.
    - Unlike L1, where small coefficients shrink faster, **L2 spreads the penalty across all coefficients**, reducing their magnitude proportionally.
3. **Impact on Coefficients**:
    - Larger coefficients (e.g., θ1=10) experience a higher penalty because θ_i^2 grows quadratically.
    - Smaller coefficients (e.g., θ5=0.1) still get penalized, but their relative contribution to the penalty is smaller.

Numerical Example

Let's use:

- Coefficients: θ1=10, θ2=6, θ3=3, θ4=0.5, θ5=0.1.
- Loss function gradients: g1=−5, g2=−3, g3=−2, g4=−1, g5=−0.1
- Regularization strength: λ=0.1

Step-by-Step Optimization

1. 1. **Gradient Updates**:
    - For L2, the gradient of $\lambda \sum \theta_i^2$ with respect to θ_i is:

$$\frac{\partial}{\partial \theta_i}\left(\lambda \sum \theta_i^2\right) = 2\lambda\theta_i$$

    - Combined gradient for each θ_i:

$$\text{Update for } \theta_i = \text{Gradient from Loss} + 2\lambda\theta_i$$

2. **Updates for Each Coefficient**:
    - For θ1:
      Gradient from Regularization: 2*0.1*10=2
      Combined Gradient: −5+2=−3 ⇒ $\theta 1_{new}$=10−(−3) =7

    - For θ2: 2*0.1*6=1.2 => −3+1.2=−1.8 ⇒ $\theta 2_{new}$=6−(−1.8) =4.22
    - For θ3: 2*0.1*3=0.6 => −2+0.6=−1.4 ⇒ $\theta 3_{new}$ =3−(−1.4) =1.62
    - For θ4 2*0.1*0.5=0.1 => −1+0.1=−0.9 ⇒ $\theta 4_{new}$ =0.5−(−0.9) =0.42
    - For θ5: 2*0.1*0.1=0.02 => −0.1+0.02=−0.08 ⇒ $\theta 5_{new}$ =0.1−(−0.08) =0.02

Result After One Update:

- θ1 10→7
- θ2 6→4.2

- θ3 3→1.6
- θ4 0.5→0.4
- θ5 0.1→0.02

Analysis

- **Larger coefficients like θ1 and θ2** are penalized more, shrinking significantly.
- **Smaller coefficients like θ5** are penalized less, shrinking slightly but not to zero.
- All coefficients are reduced proportionally and not drastically like L1.

Key Takeaway

L2 regularization works by:

1. Penalizing larger coefficients more heavily due to the quadratic term θ^2.
2. Spreading the penalty across all coefficients, reducing their values proportionally.
3. Preventing overfitting by discouraging overly large coefficients

This makes L2 regularization effective for models where all features contribute some predictive value, even if minimally.

Lasso vs Ridge:

6. **Lasso tends to do well if there are a small number of significant parameters and the others are close to zero** (ergo: when only a few predictors actually influence the response).
7. Ridge works well if there are many large parameters of about the same value (ergo: when most predictors impact the response).

Elastic Net Regularization

Elastic Net first emerged as a result of critique on lasso, whose variable selection can be too dependent on data and thus unstable. The solution is to combine the penalties of ridge regression and lasso to get the best of both worlds. sklearn.linear_model.Elasticnet

Regularization in Gradient Descent Vs Analytical Approaches(OLS)

1. **Gradient Descent**:
    - Fits in nicely to how things work in GD. Works iteratively to minimize the cost function.
    - Handles both L1 and L2 regularization directly within the optimization process.
    - Commonly used for larger datasets or complex models where analytical solutions are computationally expensive.
2. **Analytical Solutions (e.g., OLS)**:
    - Its way too roundabout; Direct solutions exist for L2 regularization (Ridge Regression).
    - L1 regularization (Lasso Regression) requires iterative optimization methods due to non-differentiability.
    - Often used for small or moderate-sized datasets where inversion of matrices is feasible.

Takeaway

- Regularization can be applied in **both analytical and optimization-based approaches**, but the implementation varies.
- L2 regularization integrates smoothly with analytical solutions (e.g., Ridge Regression), while L1 regularization (e.g., Lasso) typically requires iterative methods even for OLS.

Hyperparameter Tuning:

Strategies for Optimizing Hyperparameters

Grid search

- Divide the parameter space in a regular grid
- Execute one experiment for each point in the grid
- Simple, but wasteful

Random search

- Divide the parameter space in a random grid
- Execute one experiment for each point in the grid
- Much more efficient sampling of the hyperparameter space with respect to grid search

Bayesian Optimization

- Algorithm for searching the hyperparameter space using a Gaussian Process model
- Efficiently samples the hyperparameter space using minimal experiments

GridSearchCV()

Typically to find the best hyperparameters, one runs the models multiple times with variation and with multiple **FOR** loops. GridSearchCV function automates it all.
GridSearchCV typically prevents typing the for loops to try different parameters.
Syntax:

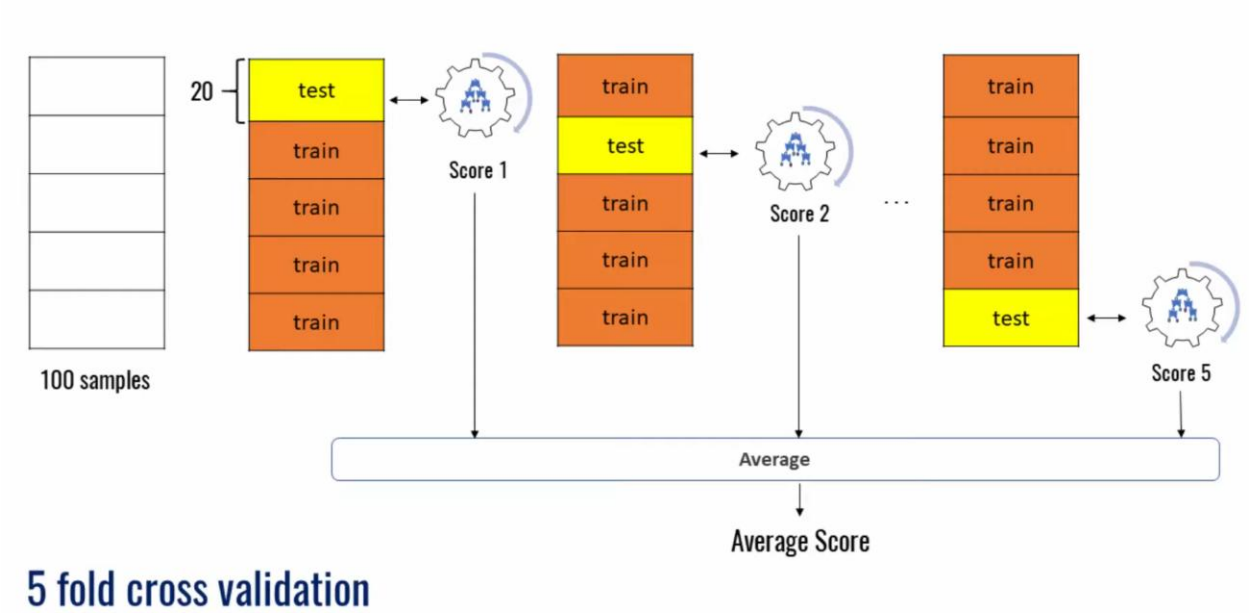
```
mod = GridSearchCV( model,
                    parameter_collection_dict,
                    cv = k)
```

RandomizedSearchCV()

GridSearch mandatorily runs every variation of the parameters hence if 5 parameters are provided that's 5! runs. What if we want 50? the compute cost is too much and besides, we might not need each and every version. RandomizedSearchCV, randomly picks parameters from those given and user controls how many runs.

K Fold Validation:

When the dataset is small and it is difficult to split the dataset 3 ways (train, validation and test) because you might not have enough data to train, validate or test. Instead, **you can use k-fold cross validation**: you divide the dataset in k parts, then you repeat the training k times, each time taking k-1 parts as your training dataset and the remaining k-th part as your validation. You record your metrics for each iteration, and then you use the average performance as your metric.



sklearn – cross_val_score(ml_model (SVM, KMeans etc), cv = k)
cv is the number of fold

Synthetic Data Generation:

1.SMOTE:

Synthetic Minority Over-sampling Technique (SMOTE)

- **Best for**: Binary classification problems, such as predicting attrition status.
- **How it works**:
  - SMOTE creates synthetic samples of the minority class by interpolating between existing data points.
  - SMOTE identifies the minority class in your dataset and generates synthetic samples for it by interpolating with one of its *k-nearest neighbors*.
  - These synthetic samples are generated by selecting a random data point from the minority class and combining it with one of its *k-nearest neighbors*.

Steps:

- For each minority sample, the algorithm selects one of its **nearest neighbors from the same class.**
- It then generates a synthetic sample by interpolating between the two points in feature space:
  - $X_{new} = X_{old} + \psi (X_{neighbor} - X_{old})$
    - xi is the minority class sample.
    - xneighbor is one of its k-nearest neighbors.
    - ψ is a random value between 0 and 1.

When to Use SMOTE

SMOTE is particularly effective in scenarios where:

- The dataset is **imbalanced**, and the minority class has too few samples.
- In machine learning models sensitive to class imbalance (e.g., **Logistic Regression, Decision Trees, Neural Networks**).

Limitations of SMOTE

1. **Risk of Overfitting**: If the minority class is highly imbalanced and SMOTE generates too many synthetic samples, the model may overfit to the synthetic data.
2. **Not Effective for All Data**: SMOTE works well for **numeric features** but **struggles with categorical data** unless preprocessed (e.g., one-hot encoding).
3. **Garbage in garbage out**: If the classes are not well-separated / well distinguished (black is black while white is white instead of all data seeming like grey) in feature space, SMOTE might generate overlapping or unrealistic synthetic samples.

SMOTE Variations

extensions and variations of SMOTE:

1. **ADASYN (Adaptive Synthetic Sampling)**: Generates more synthetic samples for minority class instances that are harder to classify.
    - Samples with more majority-class neighbors areound them are deemed harder to classify. ADASYN generates **more** of these. This focuses model learning on challenging areas, improving classification for the minority class.
2. **Borderline-SMOTE**: Focuses on generating synthetic samples for minority class instances near the decision boundary.
3. **SMOTE-NC**: Handles mixed datasets (numeric and categorical) by treating categorical features differently during interpolation.

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
```
Here, X contains your predictors, and y is the target (e.g., attrition_status).

2. Conditional Tabular Generative Adversarial Networks (CTGAN)

- **Best for**: Creating synthetic tabular data while the statistical properties of the original dataset and distributions of categorical and numerical features.
- It is particularly effective for datasets with **mixed data types** (continuous and categorical features) and **imbalanced classes**.

```
from sdv.tabular import CTGAN
model = CTGAN()
model.fit(original_data)
synthetic_data = model.sample(100)  # Generate 100 rows of synthetic data
```
`original_data` is your existing dataset.

Key Features of CTGAN:

1. **Handling Mixed Data Types**: CTGAN models both **continuous** and **categorical** columns effectively

2. **Conditional Generation**: To deal with **class imbalance**, CTGAN uses conditional sampling. For instance, it can conditionally generate rows belonging to a specific imbalanced class in the dataset
3. **Mode-Specific Normalization**: Continuous variables are transformed into a more robust representation using **Gaussian kernel density estimation (KDE)** to capture their complex distributions.
4. **Training with GAN Architecture**: CTGAN employs a **generator** to produce synthetic rows and a **discriminator** to distinguish between real and synthetic rows, iteratively improving the generator's output.
5. Captures **nonlinear relationships** between features.

3. Data Augmentation via Sampling

- **Sample rows from the dataset and add slight variations.**
- **Best for**: Small datasets where simple probabilistic methods suffice.
- **How it works**:
    - Sample rows from the dataset and add slight variations.
    - Use Gaussian noise for numerical features and random sampling for categorical features.

```
import pandas as pd
import numpy as np
def augment_data(data, n_samples):
    synthetic_data = data.sample(n=n_samples, replace=True).reset_index(drop=True)
    for col in data.select_dtypes(include=['float', 'int']):  # Add noise to numerical columns
        synthetic_data[col] += np.random.normal(0, 0.01, size=n_samples)
    return synthetic_data
```

augmented_data = augment_data(original_data, 50) # Generate 50 synthetic rows

4. Faker Library for Randomized Data

- **Best for**: Quick data generation for non-sensitive data, e.g., creating mock datasets.
- **How it works**:
    - Faker generates random but realistic data.
    - Combine it with probabilistic distributions for better control.

```
from faker import Faker
import pandas as pd
import random
faker = Faker()
synthetic_data = []
for _ in range(50):  # Generate 50 rows
    row = {
        "employee_id": faker.random_number(digits=5),
        "gender": random.choice(["Male", "Female"]),
        "years_of_service": random.randint(0, 30),
        "market_salary_percentile": round(random.uniform(0, 1), 2),
        "performance_rating": random.randint(1, 5),
        "attrition_status": random.choice([0, 1]),
    }
    synthetic_data.append(row)


synthetic_df = pd.DataFrame(synthetic_data)
```

Which Method Should You Use?

- **Balanced Data**: Use SMOTE for class balancing.
- **Complex Patterns**: Use CTGAN or customized statistical methods.
- **Quick Augmentation**: Use data augmentation or Faker.

Neural Network

3Blue1Brown: https://www.youtube.com/watch?v=aircAruvnKk
https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

Machine Learning / Deep Learning Tutorials for Programmers playlist:
https://www.youtube.com/playlist?list=PLZbbT5o_s2xq7LwI2y8_QtvuXZedL6tQU by DeepLizard

What NN is all about?

1. NNs are about breaking down all input into smaller portions / abstractions to analyze
   - A Neuron activates if input "analysed" is high or deactivate if low
2. Then the smaller portions / abstractions are joined together to realize a bigger pattern.
3. From these bigger patterns, NN should have learned to decide as per their purpose like tell when the number is: 3.

- Although we create these neurons and it holding values, the way data is processed through a network is with numbers in matrix forms when many of these calculations are infact done simulataneously / parallely.

1) NN is about breaking down the input into smaller portions to analyze. Eg: If input are pixel values of an image – then you can imaging NN breaking it into different sections of the image and looking at the input only in that small region. Like the small boxes in Fig 1 below.
The "Analysis" is basically (application of a weight and bias) to see if neuron activates when input studied is high or deactivate if low.

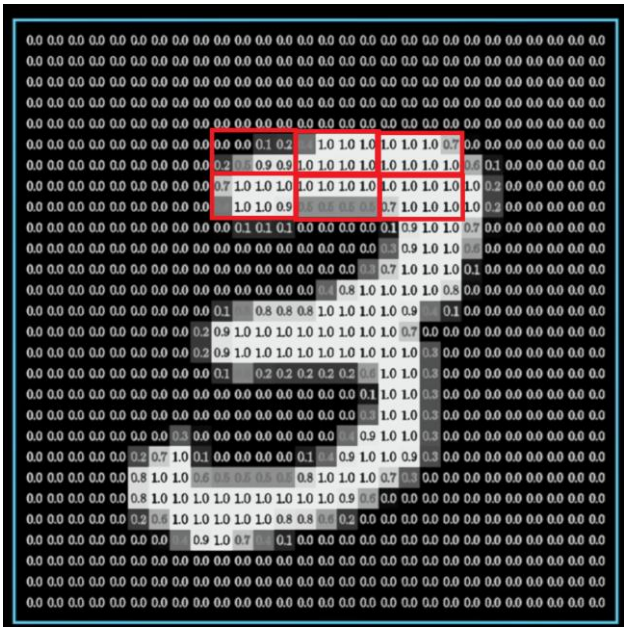


Fig 1: Breaking down input into smaller combinations.

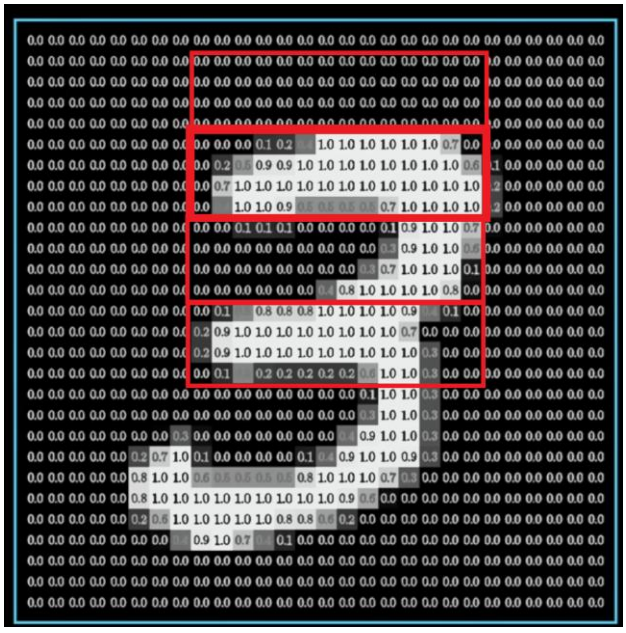


Fig 2: Combining patterns to realize bigger patterns.

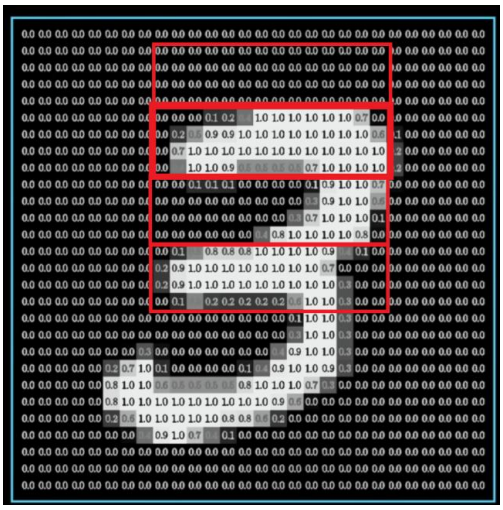
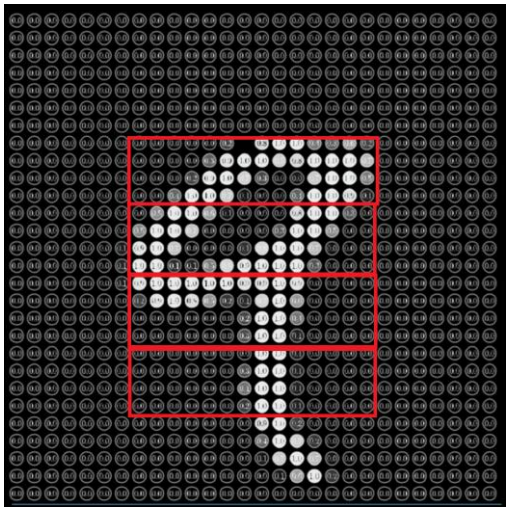
> Each box above is a neuron in Hidden Layer 1. **> How does a neuron in HL1 focus only on its box / only few inputs instead of all: WEIGHTS.** The weights for input from cells within box are high, amplifying them, while weights for other input cells are zeroed by the weight.	**> Each box above is a neuron in Hidden Layer 2.** **> How does a neuron in HL2 focus only on its big box: WEIGHTS.** > The values speak for themselves in that high/activated/white cells in previous layer when combined in the next layer makes the neuron in HL2 activated or not. The neuron for first box above will be 0 and so unactivated when neuron2 responsible for box 2 might activate.

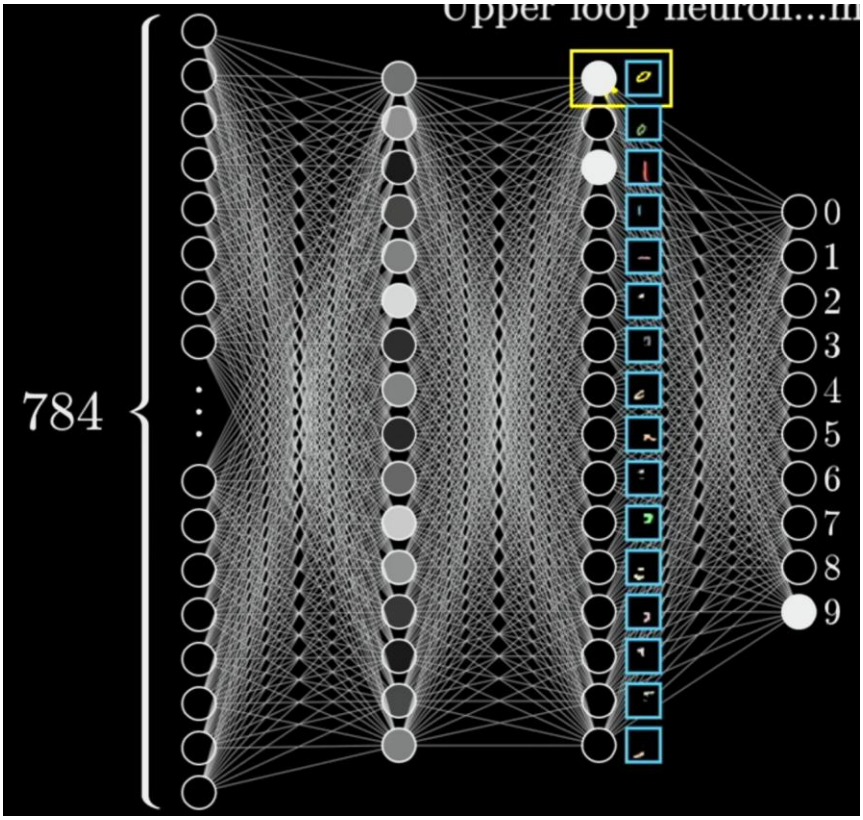
2) NNs then combine the smaller boxes to bigger boxes to slowly picking bigger patterns in the data as we move to the next and next layer.

3) Decision / Output Layer: Decision / Output layer neurons are like Senior Management of a company each with their own objectives to achieve. All previous layers are worker nodes that divide the work and combine to show different patterns recognized. The output layer picks only the patterns in the last hidden layer that it is interested in and focuses on few and activates when they activate and deactivate if they don't.

Thus, from the final numbers in the output layer, NN see which neuron fired and predicts the number: 3.
The neuron in final decision layer is slightly different and it only sees if specific portions in previous layer activate or not.

- How does it decide to focus only on few portions and not other portions: Weights

<table>
<tr>
<td>  </td>
<td>  </td>
</tr>
<tr>
<td colspan="2">Decision Layer Neurons
> In the decision layer, there is a neuron for each number. Each neuron is focused only on the activations of sections that are specific to its number.
> The weights help it to cutoff sections that it is not interested in</td>
</tr>
</table>



Output layer neuron each have an objective set and they focus on only few neurons in the last Hidden layer and activated when its neurons fire and deactivates if they don't. How do they focus on only few neurons and silence the rest: weights and biases

Each neuron is doing an analysis which is creating a linear combination: multiplying with a weight and applying bias. The weight and bias are basically control variables that allow the NN to control what to focus and what not to. The control variables also get optimized during the training and feedback (back propag) process of a NN.

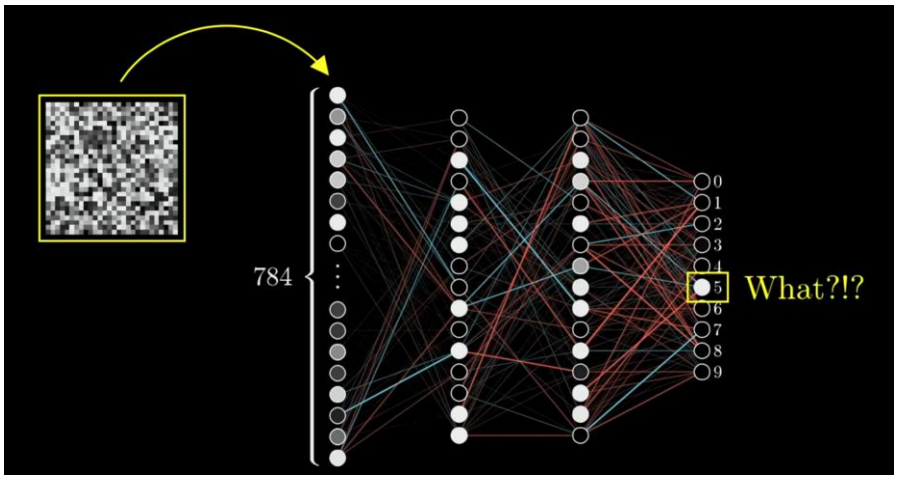
The training and feedback is like trial and error way to an answer as in trial of different weights, find the error and then correct the error though back propag.

During the training and feedback runs, the decision layer neurons which have the answer already stored (supervised learning) help the NN to give feedback to how the weights of previous layers should change such that they focus on the activations of the right set of neurons in the previous layer.

Do note: that each output neuron will train only specific portions of the network and its not like the entire network works to help it answer its question. How? Weights help it switch off unwanted sections including sections that are relevant to other decision neurons.

Training and Feedback: mentioned about is all abt finding the error and back propagation; sending the feedback back through the network so weights get updated.

Summary: First, NN don't break things down into boxes and combine boxes systematically (one next to each other etc) as mentioned above. The neurons could be all over the place and only during back propags are the weights updated and each layer forced to pick a specific pattern – a pattern which might not make human sense but definitely effective (back propag ensures it is made effective).

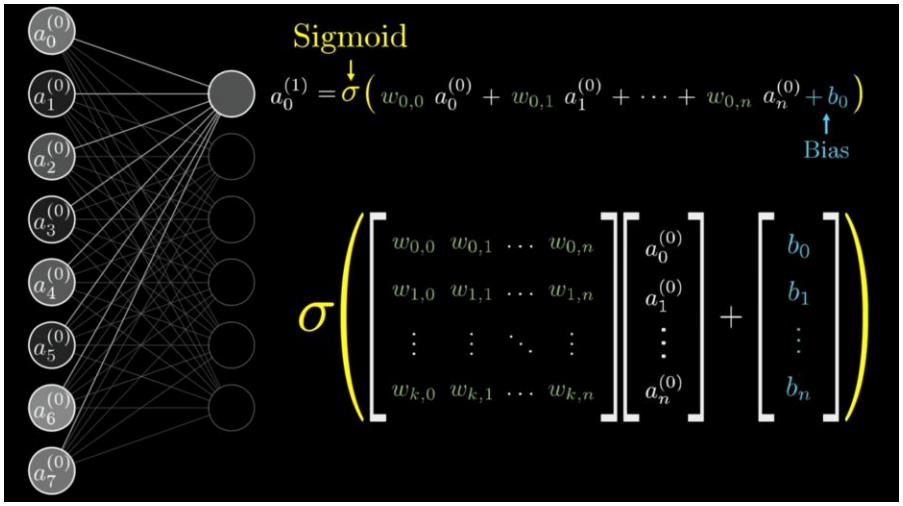


Eg: When a random picture with random pixel values was passed thro the NN, it confidently guessed as 5. This is what we mean by the patterns and boxes it picks are not systematic but it does its job within given constraints and input well.

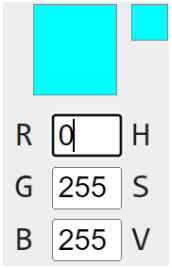
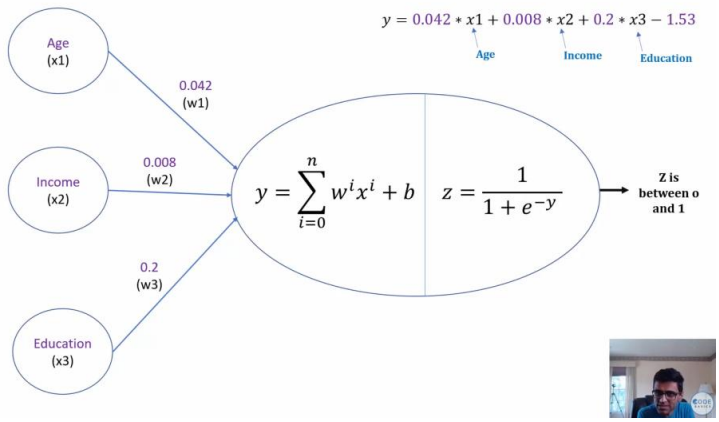
This is why NN can do a lot of things it is known to do, it just breaks down the input into smaller sections, combines them and during training/feedback learns to make these sections and patterns useful. This abstract way of breaking things down and bringing it together is why it works for different applications.

Linear Algebra Layer:

- Although we create these neurons and it holding values, the way data is processed through a network is with numbers in matrix forms when many of these calculations are infact done simulataneously / parallely.



Imagine, NN has to guess if a given a color, R:G:B proportions, it is Aqua color(0:255:255) which is known or set at output layer of NN.
Proportion of Red, Green and Blue are entered in the input node layer = 3 nodes.
Here we simply enter different combos for red, green and blue as inputs like 0, 155, 255 for each color.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</table>

- A weight gets applied and then squashed to a smaller number and passed along.

Thus the weights and the squashing ends of trying different combinations of R:G:B and different weights for each R, G and B input. While the final output node has the answer which gives feedback on how wrong the squashed number is thus judging the input and weights while the combination which is closest say: 0:255:250 and weights that promoted this combo will show a lower error than others. The size / magnitude of error is sent back to each weight during back propagation and so NN network learns by reinforcing good weights/inputs while others penalized. With Gradient Descent the weights are also updated and after many epochs the NN learns to get it right.

https://www.youtube.com/watch?v=_N5kpSMDf4o&t=39s

Neuron: A cell that is a function (aka a formula to be applied) that spits out a number that is held by the neuron. The number indicates its "activation". It is high and thus activated and low when not activated.

In fact, the entire Network is like a function that takes in values and spits out numbers.

Weights and Biases

Weights are an important control variable that assists a NN to selectively process an input value. Weights are usually number between -1 and 1.

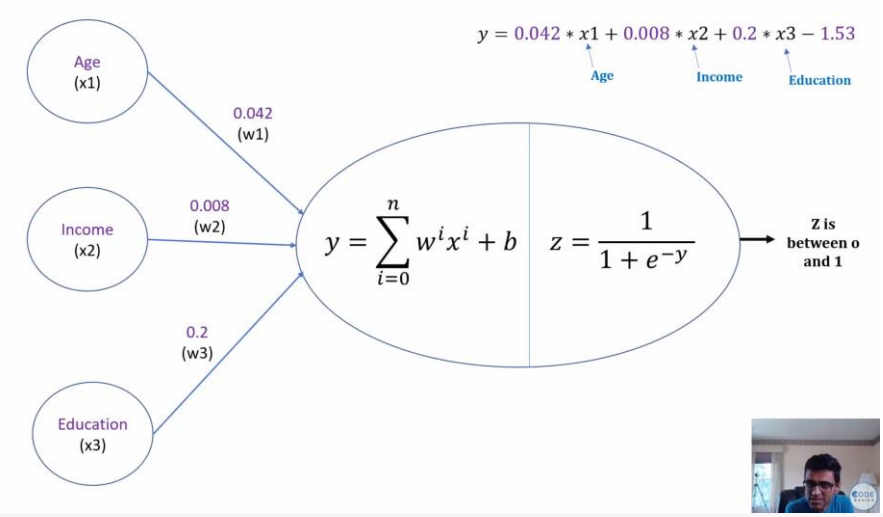
We train neural networks with gradient descent, the training can proceed faster if the weights stay in the vicinity of where the activation function changes rapidly, i.e., close to 0.
The weights of a neural network are typically initialized with a mean of 0, i.e., some weights are negative and some are positive, but they are in general between -1 and +1, close to 0.

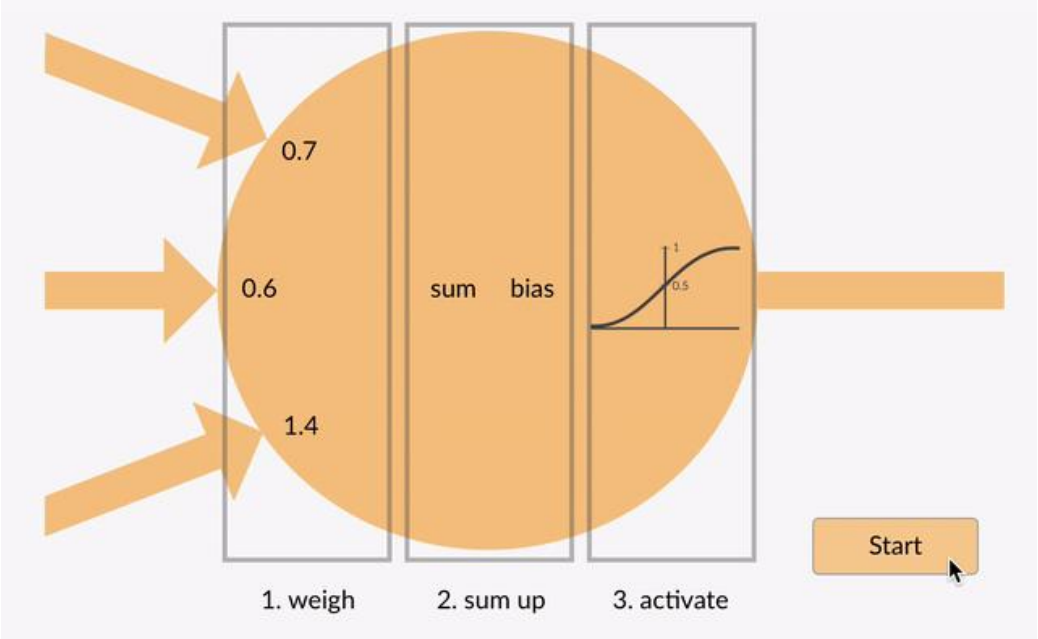
Lastly, Other important components like

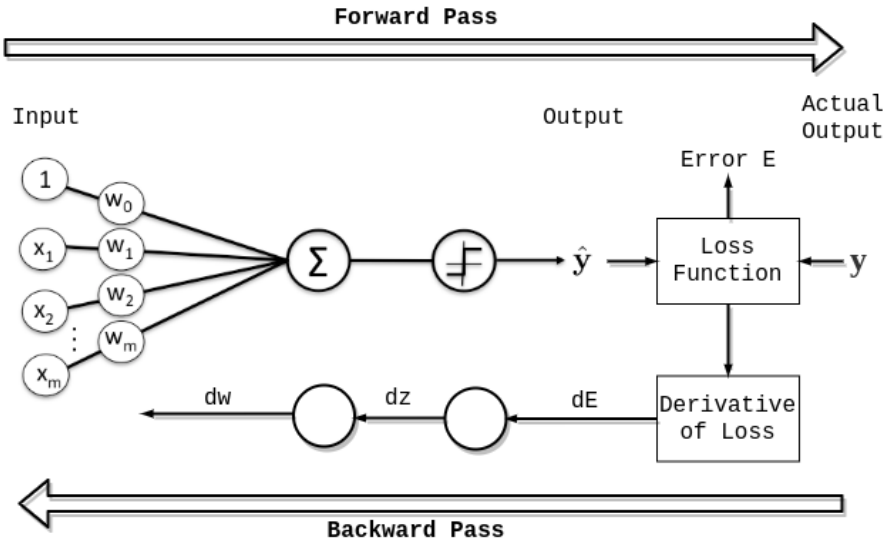
1. Activation function is to get a final comparable number as output in a neuron that indicates if it fired or not.
2. Bias is a constant number that is added or subtracted to the weighted sum either to set a threshold for the value or even give an artificial boost.

Neuron
https://www.youtube.com/watch?v=VhRtaziEWd4&list=PLeo1K3hjS3uu7CxAacxVndI4bE_o3BDtO&index=3

Neuron tries some weights or coefficients to describe the relation and then uses a classifier 'Activation Function' to decide to fire/activate or not.







Neuron has two parts the

8. stem where weights are applied and
    a. weights are like coefficients calculated in a multinomial regression

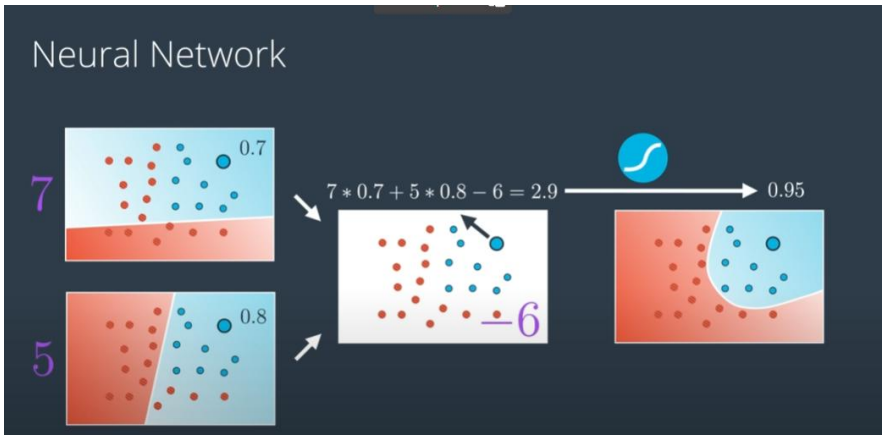
9. the nucleus – Sum up and activation
    a. Sum up:
    Values from each stem are sumed up and a bias is applied.
    What is the function of bias?
    1) Bias decides if certain low values should be included to trigger a "fire":
    Here the Bias is a +ve value added to whole sum.
    Eg: if Bias is a +2 and the Sumed up value happens to be -1.35 then +2 makes it positive and if ReLU was the activation this forced +2 included the small negative value of -1.35 to be included in group of values that will trigger neuron to "fire"
    2) Bias decides if certain early positive numbers should be excluded from triggering a "fire":
    Eg: if Bias is -10 and the sumed up value happens to be 7 then -10 makes it -3 and so neuron does not "fire". Thus a minimum threshold is being set here for neuron to "fire"
    Bias thus helps to be more inclusive or more exclusive.
    b. (activation) fires or not based on the activation function
    activation function helps classify the incoming numbers which is used by the nucleus to decide to fire or not

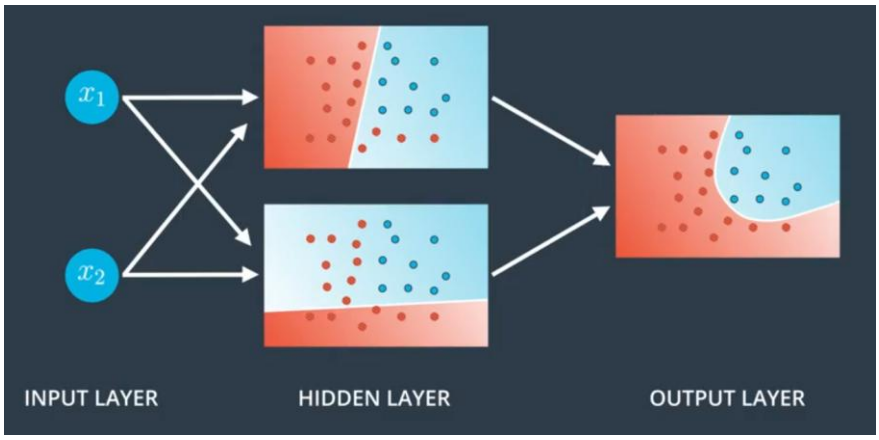
NN patterning Non-Linear relationship:

Watch Udacity Video – Sec2 NN Intro – Video 4

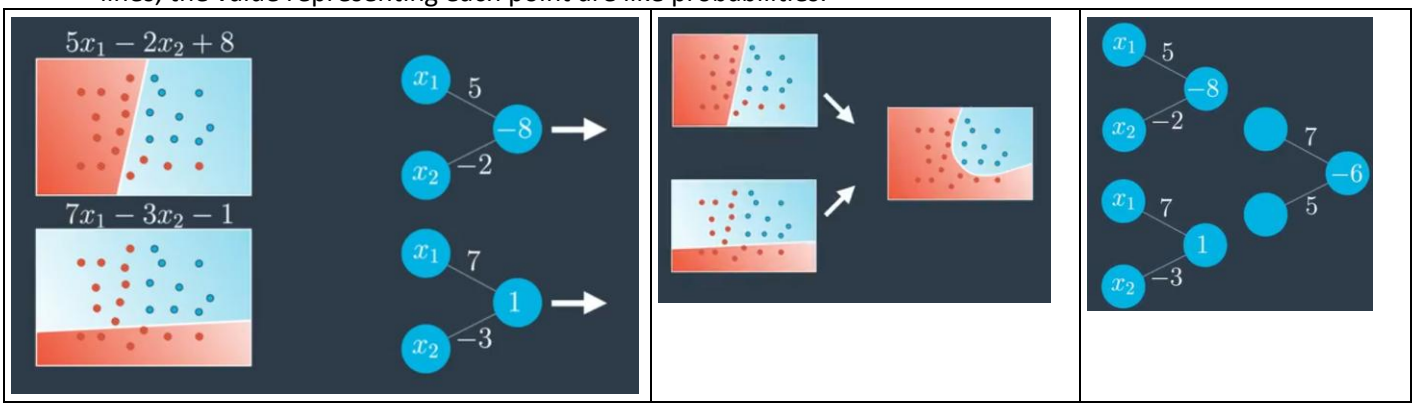
How do NN captures the pattern in data?
What really happens between two hidden layers in NN?
How is the non-linear relation brought out and remembered by NN?
How is NN better (by patterning the non-linear relation) than Linear regression or similar linear methods?

Simply, NN s combine linear relationships (linear eqns) found in the initial layer into a non-linear eqns using activation functions.

- First, It is very important to understand that after the first hidden layer and before the second hidden layer, the **neuron outputs are basically like probabilities** since the activation functions make it so.
- 



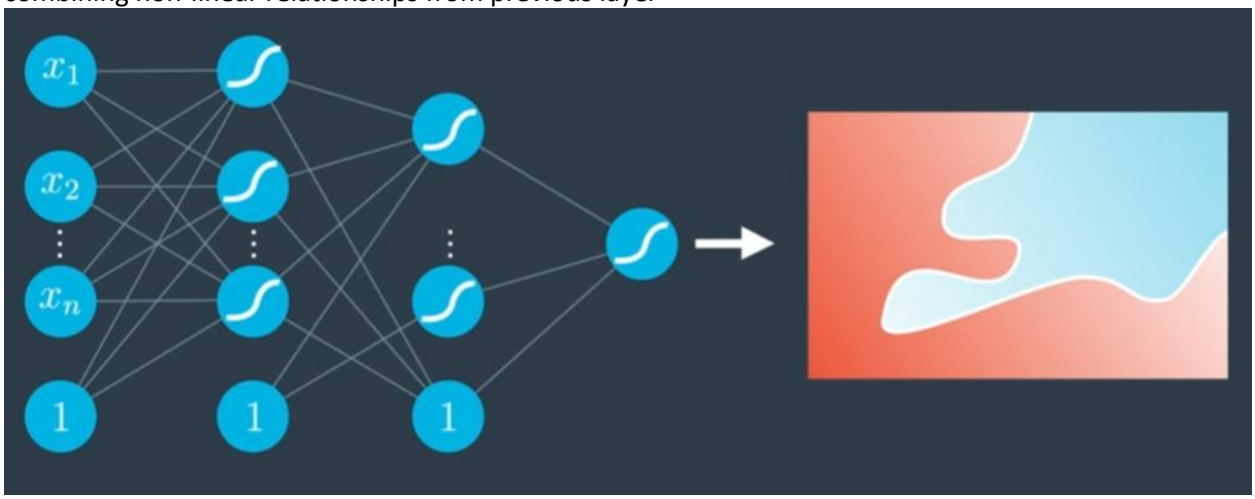
- As in picture above, imagine that two linear relationships developed at two nodes and based on these linear lines, the value representing each point are like probabilities.



- When these (data of twin points) are sent over to the next layer, the inputs are weighted and biased and then scaled to another probability like value. When all the twin points data are sent through, you end up with a new probability distribution for whole data set which depict a non-linear relationship.

Data Values are converted to probability values between the layers and by combination of the initial linear relationships at the next layer, non-linear patterning of the data is unearthed by the NN.

Keep doing above by increasing hidden layers and NN can generate very complex non-linear relationships by combining non-linear relationships from previous layer



NN is good at picking out the Non-linear relationship input and expected output. NN divides the building of the non-linear curve among it multiple nodes and layers where each takes care of one part of the curved non linear

relationship and after many epochs the curve is finalized to its complex curved form. The freedom to separate the work between different nodes and stitching it together is where other simpler methods fail since they don't have back-propagation and stuck in what they do.

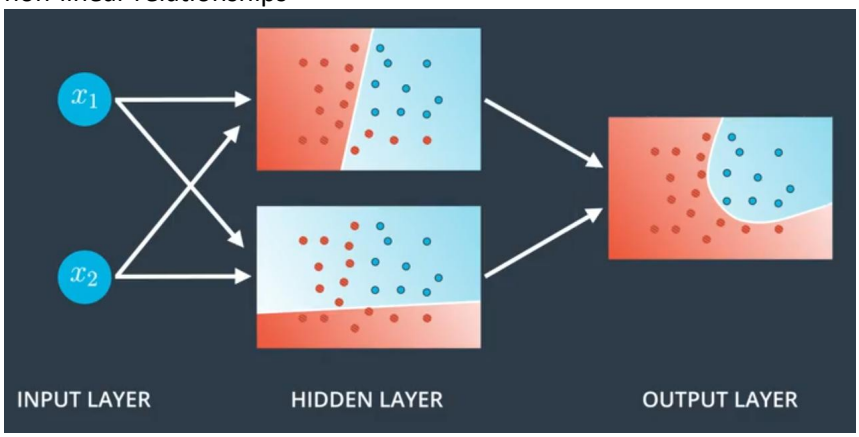
1) https://www.youtube.com/watch?v=ILsA4nyG7I0 – Brandon Rohrer
2) https://www.youtube.com/watch?v=aircAruvnKk - 3blue1Brown
NN are good for learning different patterns.

NN Workflow – Single EPOCH:

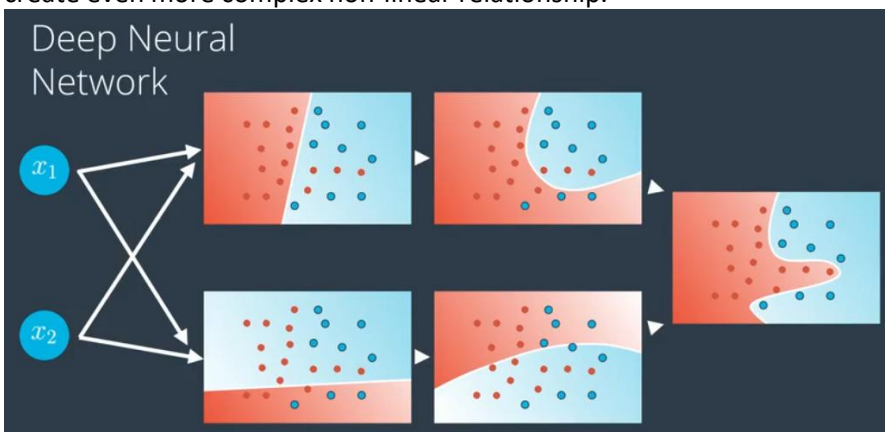
A network of layers where each layer has multiple node/neurons.

1. At the First Hidden Layer, each neuron applies a coefficient aka weight to an input factor/variable. (each neuron in a way is finding a linear relationship between the inputs in same design as Linear regression → y = mx +c

2. At the next layer, which if is Output, the initially concocted linear relationships are combined to create a non-linear relationships



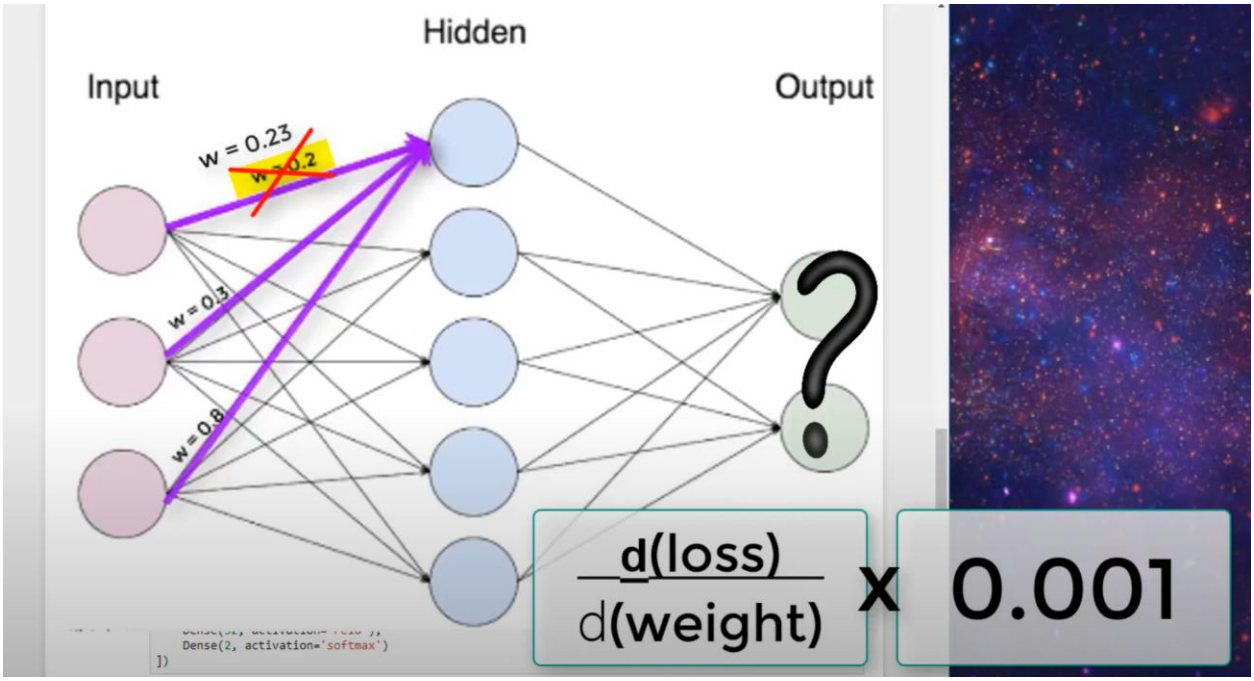
3. If the Next layer is another Hidden Layer, then non-linear relationships are formed, which later get added to create even more complex non-linear relationship.



4. When data reaches Output layer, this is one FeedForward and next is back propagation.
Output layers have labelled data so error is found and then gradient descent is found for each weight. Learning Rate is applied which reduces the change in weight to a small number. Weights by themselves are small numbers or even fractions 0<w<1 so LR are even smaller like 0.01 < LR< 0.001 etc.

   Back Propagation: mechanism of sending relevant feedback (change in weights) to respective nodes. The **matrix form of passing weights and doing Linear Algebra, helps in easily targeting each weight individually**.

5. Using gradient descent, the weight is updated such that a neurons that got it right (low error during first/previous epoch) are boosted and those wrong (high error during first/previous epoch) are reduced in the right direction. Thus each node and its weight gets updated. This is one epoch.



After first epoch – updating weight based on error

6. The same data is fed forward again and now with updated weights should give lower error. Cycle is repeated, once data is run through (an epoch) -> error of output is found -> gradient descent for each weight is found and back propagated -> weights updated -> another epoch is run. Thus the network learns (basically updating its weights/coeffs with reference to output)

NN BackPropagation

Backpropagation is calculation of the negative gradient of the cost function which tells by ho much each weight and bias must change for the best decrease in cost/error.

Backpropagation is about calculating the gradient for each node with respect to the relevant weight and its influence on the final error.

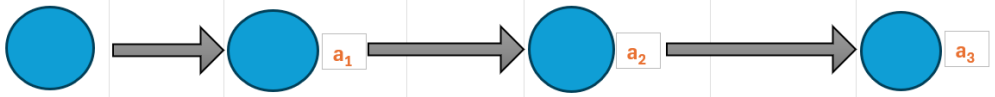
Backpropagation is a complex and tedious business since each weight and activation had an impact and so to back trace and include all of in relevant way is complex. Backpropagation does it effectively using the Chain Rule which formulaically includes all the relevant factors and estimates the relevant gradient custom for each node.

Below is an example of a back propagation.
Situation:

- Forward Pass: Input = 1, Assume all weights = 0.5
- Activation Function = Sigmoid

- Back Propag: Error = 0.1, LR = 0.01
  - Partial Derivative of Activation Func, Sigmoid = Ax * (1 – Ax) = Activated Value * (1 - Activated Value)
  - Gradient = PD * BPInput → (Partial Derivative * Appropriate input coming thro Back Propag)

- BPInput is the Error when Back Propag is started and after that, the Gradient * Weight from previous step becomes the BPInput

					G2 is dependent on G3 * w3 from previous layer	With LR and small Gx, the udpate to weight is super small		**Back Propag:**
			PD1 = 0.235		PD2 = 0.244		$PD_3 = a_3 * (1 - a_3)$ = 0.5717 * 0.1 = 0.244	Formulas: **Gradient, Gx = PDx * $Input_{BP}$.** $Input_{BP}$ is E at the last node and Previous Gradient for earlier nodes
			G1 = PD1 * w2 * G2 = 0.235 * 0.5 * 0.002 = 0.0003		G2 = PD2 * **w3** * **G3** = 0.244 * 0.5 * 0.024 = 0.002		G3 = PD3 * E = 0.244 * 0.1 = 0.02	**Partial Derivative, PDx = a_x * (1 - a_x)** As applicable to Sigmoid formula. Diff Act. Func will have diff PD formula
			G1 = 0.0003		**G2 = 0.002**		**G3 = 0.02**	E = 0.1 , LR = 0.01
		$w1_{new}$ = 0.499997		**$w2_{new}$ = 0.49997**		**$w3_{new}$ = 0.4997**		**$w_{new} = w_{old}$ - LR * Gx**
								
	Input		Hidden 1		Hidden 2		Output	
Feed Forward:		**w1 = 0.5**		**w2 = 0.5**		**w3 = 0.5**		
a_x = Neuron's final output at a layer. value after activation func is applied on incoming values	Input = 1		$a_1 = \sigma (0.5 * 1)$		$a_2 = \sigma (0.5 * 0.622)$		$a_3 = \sigma (0.5 * 0.577)$	
Act. Func = Sigmoid **$a_x = \sigma (w_x * Input_{FF})$**			$a_1 = 0.622$		$a_2 = 0.577$		$a_3 = 0.5717$ **(Final Output)**	
$Input_{FF}$ is prev. layer output								

Chain Rule in BackPropagation

Backpropagation uses the chain rule to compute how the weights in each layer affect the final loss.

The whole purpose of Backpropagation in NN is that we want the gradients (derivative) of the loss function (error) with respect to the weights. d(error) / d(weight)

First, Each neuron is a composition of :

1. A **weighted sum of inputs** (z=wx+b).
2. An **activation function** applied to that sum (a=σ(z)).
3. Output to next layer or the final **Loss** function of NN.

We need, d(error) / d(weight) but the weights do not directly affect the loss, before which:

- weights affect z → z affects a → a affects Output/Loss.

Chain Rule allows us to unpack these when calculating: d(error) / d(weight)

$$\frac{\partial \text{Loss}}{\partial w} = \frac{\partial \text{Loss}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Chain Rule Definition:
In differential calculus, the Chain Rule is a formula used to find the partial derivative of a nested functions or composite function.
If u(x) = t is a simple function then v (u(x)) is composite/nested function. Now if f = v(u(x)) then differentiable df/dx = df/dv * dv/dt.
Chain Rule applies for Partial derivatives as it does for Total derivatives.
Applying Chain Rule, we can expand d(Error) / d(weight) as

$$\frac{\partial \text{Loss}}{\partial w} = \frac{\partial \text{Loss}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

In fact, Each term above represents taking into account each dependency in the chain of dependencies:

- d(Loss) / d(a) : How much the Loss changes with respect to the activation func.
- d(a) / d(z) : How much the activation changes with respect to the pre-activation weighted sum value.
- d(z) / d(weight) : How much the pre-activation changes with respect to the weights.

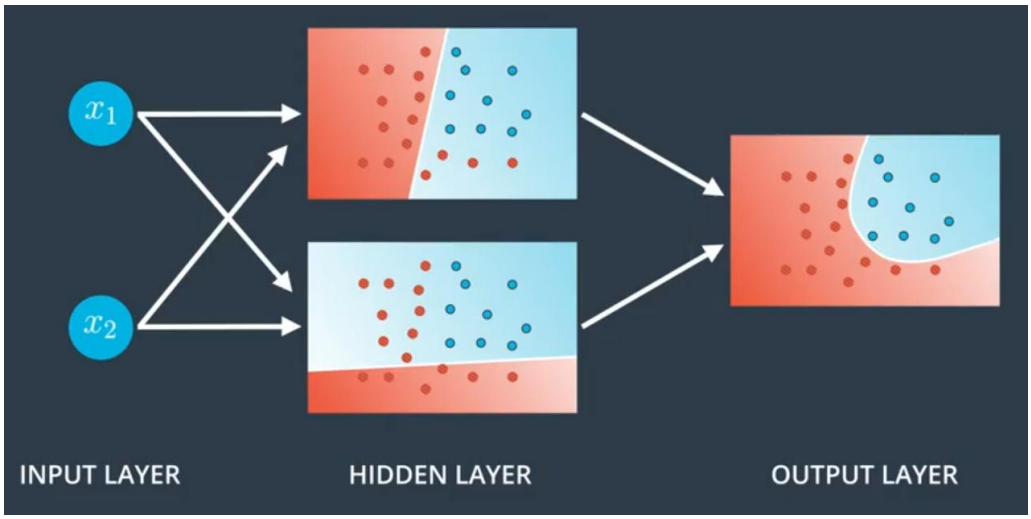
This is the essence of backpropagation and why the chain rule is fundamental to make it work!

NN and the Architecture FAQ:

What happens if we have multiple neurons at input?
More hidden layers?
More Output neurons?

Watch Udacity Video – Sec2 NN Intro – Video 7

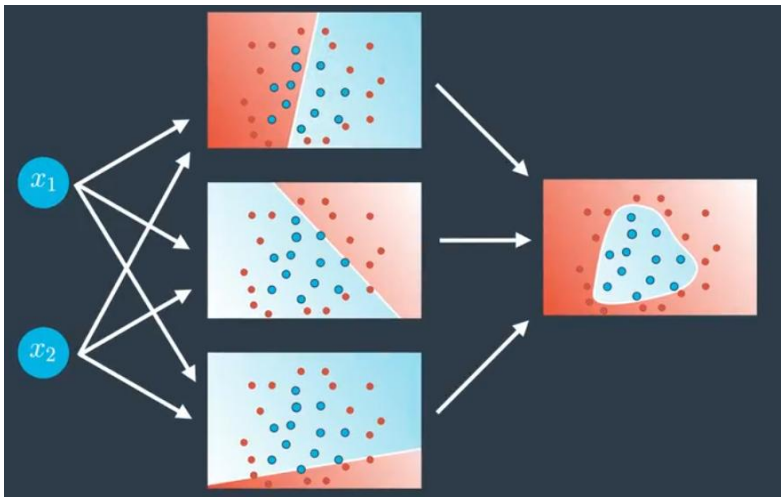
Simple Architecture:



1. First Hidden layer create linear relationships
2. Output layer combines the two to create a non-linear relationship

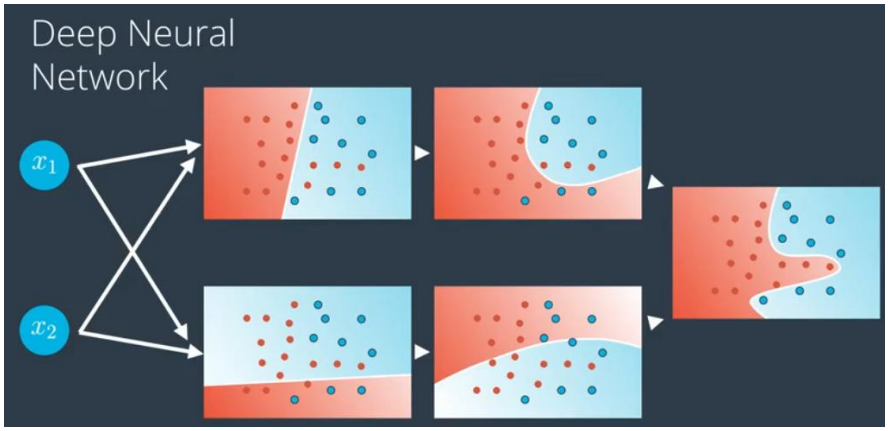
Larger Hidden Layer:

More nodes in Hidden layer

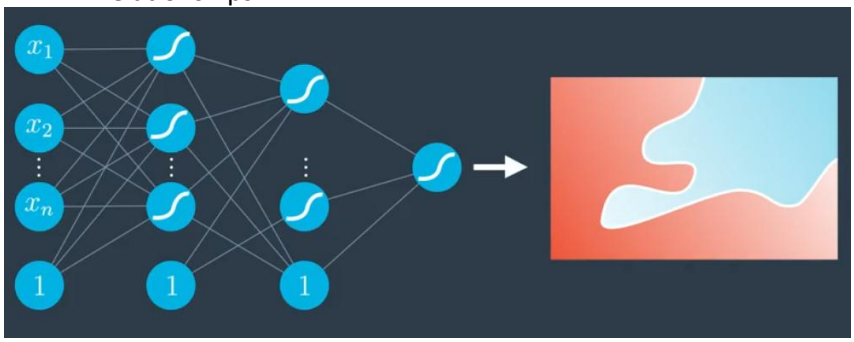


- More nodes in hidden layer → More linear relationships → Final Non-Linear Relationship is more complex

More Layers:



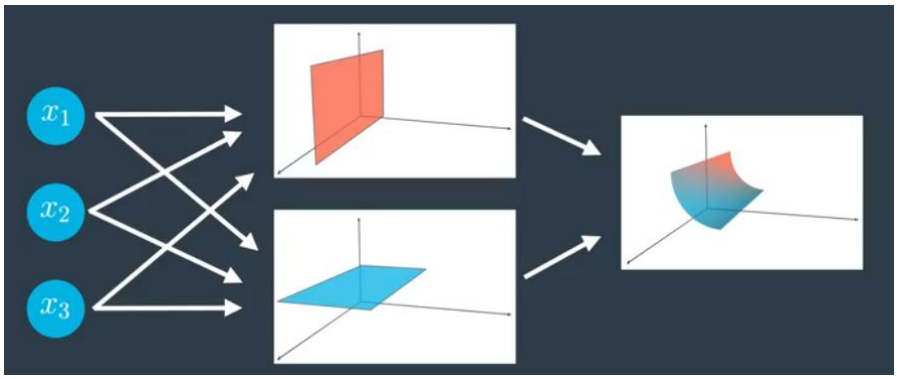
- More layers means non-linear relationships get combined to form even more complex non-linear relationships



- More layers could be added and more complex relationships be identified

More Input Nodes:

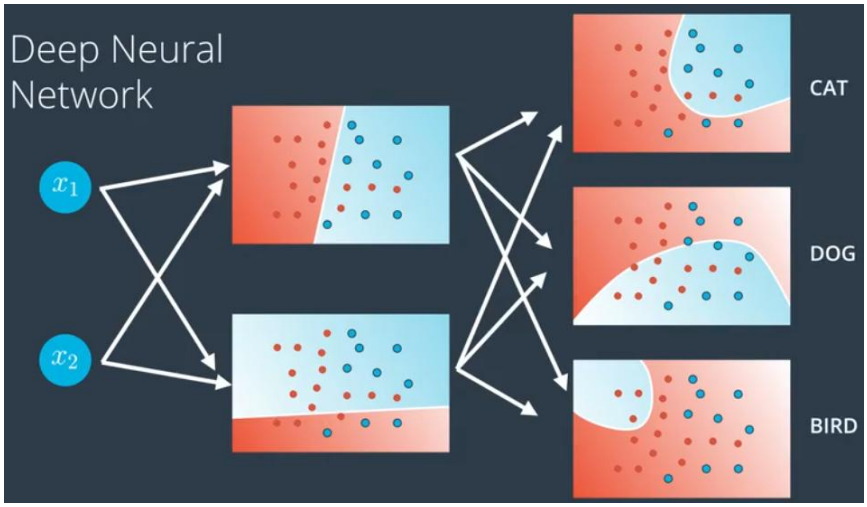
Input node number depends on data – It should need n-dimensional space. 2D means 2 nodes, 3D means 3 nodes etc.



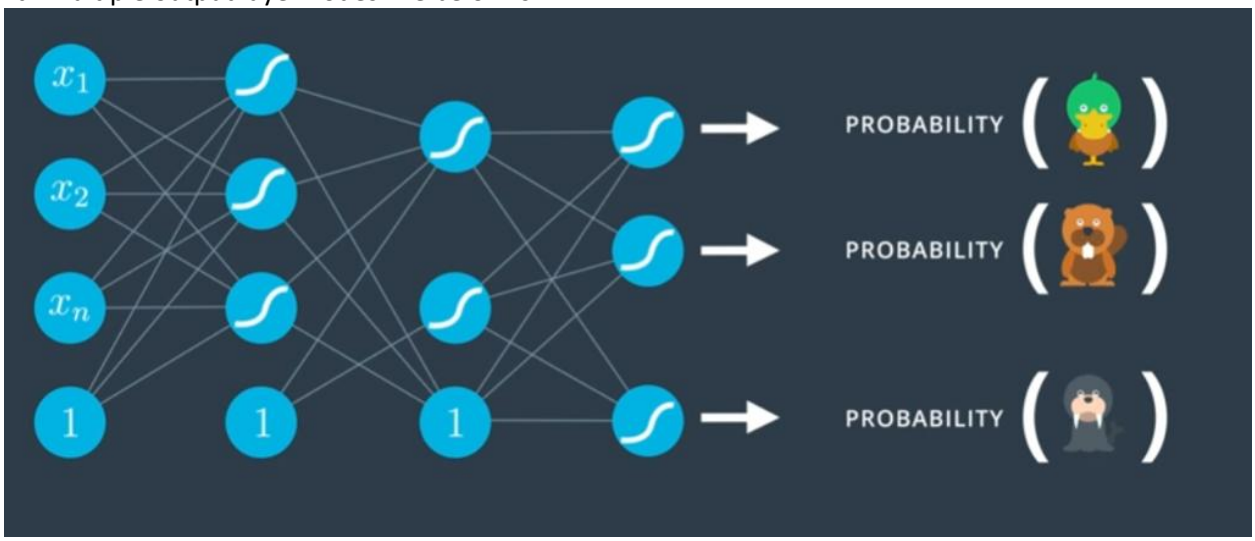
- More Input means the Input itself is in higher dimension like 3D coordinates.
- First Hidden layers approximates – 2D relationships while Output layer brings it together in 3D

Many Output Nodes / Multi- Class Classification NN:

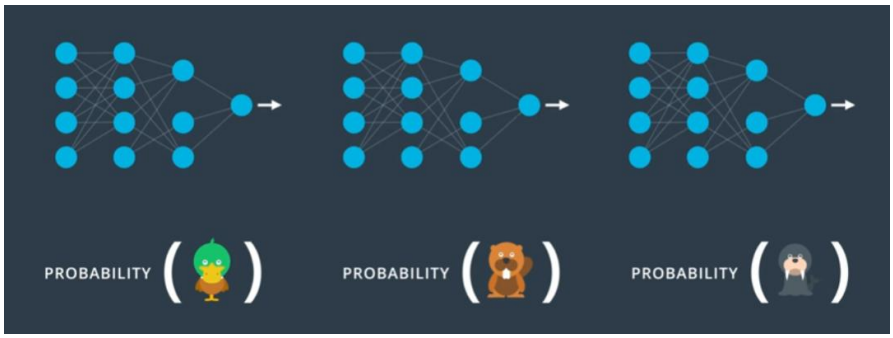
Output Node Number depends on the case – If classifying 3 classes then 3 nodes etc.



NNs with multiple output layer nodes like below is



Is actually, many NNs which would have only classified one object like in previous NNs merged into one, since initial layers are doing the same job if they are in separate NNs



Data Types and I/O for NN

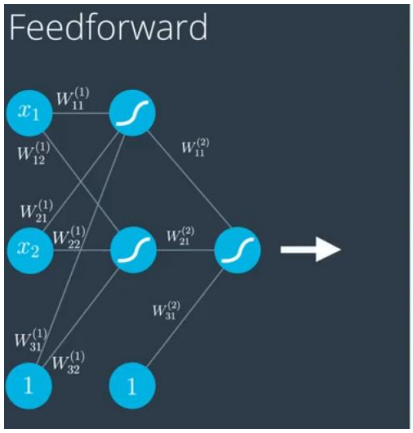
NeuralNetworks work with only numeric data so categorical data must be converted to numeric form (encoded).
One Hot encoding is popular but still naïve while word-embedding as done in NLP is latest and better approach for NN. link1 link2

Recommended Preprocessing:
Data normalization is an important pre-processing step for neural networks. The activation functions that are normally used in neural networks (sigmoid, ReLU, ...) have the highest rate of change around 0 – so we should get the values themselves close to 0 since multiplication of numbers close to 0 remain close to 0.

The weights with which the values are multiplied will already be close to 0.
Therefore, our transformed data will contain both negative and positive pixels with mean 0 and standard deviation 1. Sometimes you'll see an approximation here, where we use a mean and standard deviation of 0.5 to center the values.

NN – Math/Linear Algebra Layer

Watch Udacity Video – Sec2 NN Intro – Video 8
All of above notes are from the Visual layer or Architecture layer. It is purely from architecture perspective as in how many input nodes, layers and the output.



$$\hat{y} = \sigma \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix} \sigma \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

The way the data is handled is in matrices in the Linear Algebra Layer

In the Linear Algebra Layer:

1. The inputs are assembled in vector form [x1,x2,1]
2. Weights (randomly initialized in first run) are also in matrix form

3. Activation func – sigmoid is multiplied

$$\sigma \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

4. Actually, all the layers are assembled in one go and not one layer at a time as pictured or explained. Yhat is thus directly calculated using Linear Algebra.

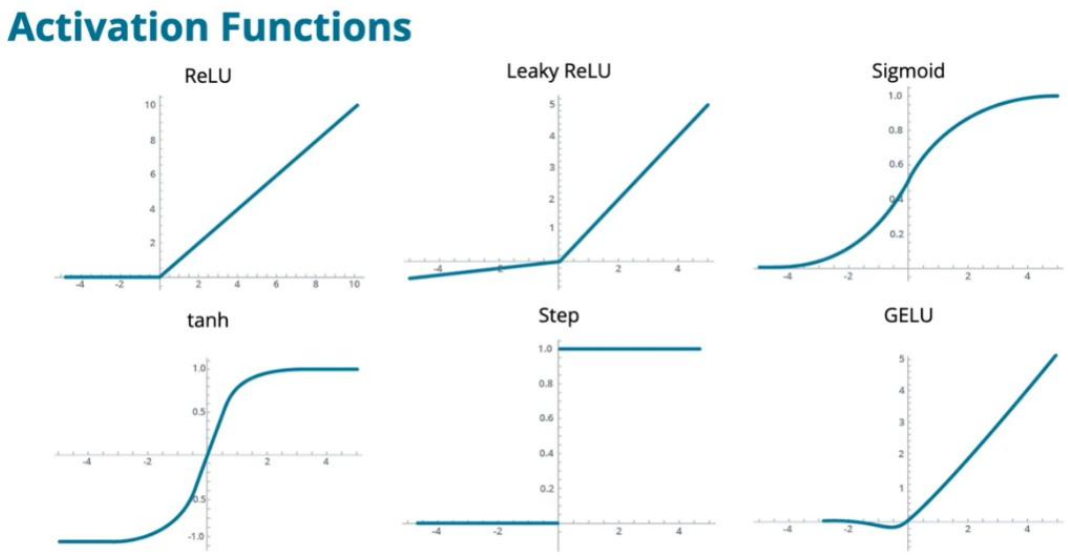
$$\hat{y} = \sigma \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix} \sigma \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

$$\hat{y} = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

Or in even simpler Notation:

5.

Activation Functions



The purpose of an activation function is to scale the outputs of a layer so that they are consistent, small values. Much like normalizing input values, this step ensures that our model trains efficiently!

Activation functions should be:

- Monotonic – value of y is increase when going from left to right in the graph
- Differentiable – can do math differential and its continuous

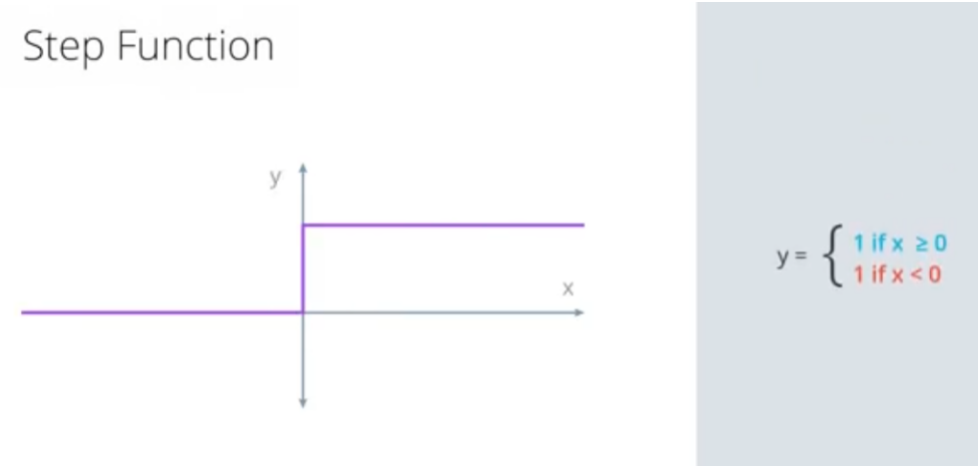
- Nonlinear - not just a straight line else it is just good for linear relationships
- Close to the identity function at the origin

We can loosen these restrictions slightly. For example, ReLU is not differentiable at the origin.
Others, like monotonicity, are very important and cannot be reasonably relaxed.

What factors can help you decide what activation function to use?

1. Ease of computing gradient
2. Value at origin - 0 – Expected to be close to 1
3. Whether or not activation func is bounded from above or below – This is the cause in case of Vanishing or Exploding gradient issue

Step Function:



Simple function that check if value is less than zero or above.

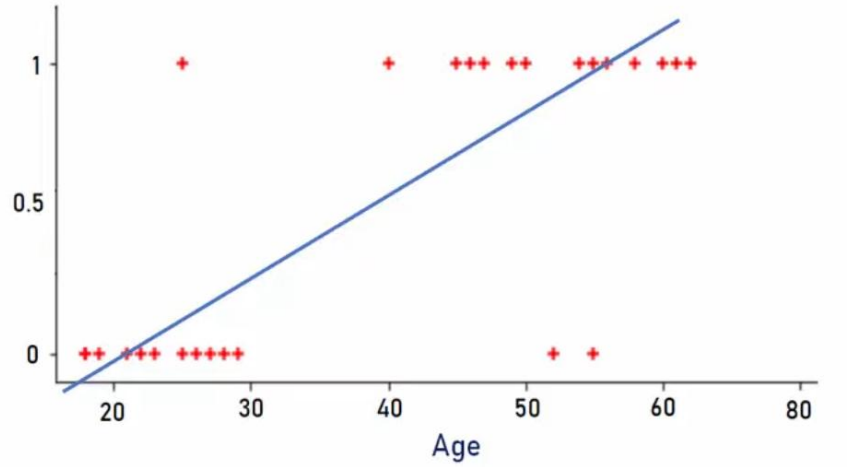
First, Regression:

First, understand Linear Regression before what and why of the Activation functions.
Example: Age of people is input and we have to guess if they will buy insurance or not

Step1: calculate the regression line with the data. thus you get a y = mx + b

$$y = m * x + b$$

Age



The Regression equation is below:

$$y = 0.042 * x - 1.53$$

Age

Now if you put in an age, the eqn spits out a number like below.
Notice, the younger's result are negative while older are positive.
So we find when results gets to zero and make it the cut-off age.
The number seems be between 35 and 38. I did some tweaking to get to zero and 36.429 seems to be a good bet.

Age	Linear Reg Eqn Result
15	-0.9
20	-0.69
25	-0.48
30	-0.27
35	-0.06
36.5	0.003
36.429	1.8E-05
38	0.066
45	0.36
50	0.57
68	1.326
72	1.494

Notice: The linear reg eqn in a way scaled the input (15 to 72 range) to (-0.9 to 1.49 range). Basically scaling down input to comparable numbers

Note: Linear eqn being linear is **not helping** to divide the data into two sets towards 0 (no insurance) and 1 (bought insurance)

Refer MATH Key concepts.docx (also below) file for more explanation/comparison of sigmoid.ReLU

Sigmoid / Logit

ML: Logistic Regression, Neural Network, Neuron, Activation function

Sigmoid function converts input into range 0 to 1

Takes bunch of numbers and converts to 0 to 1 which is like normalization/percentile calc so why not just that. Sigmoid has slightly bigger range, -100 to 100, where -100 is 0 and 100 is 1.

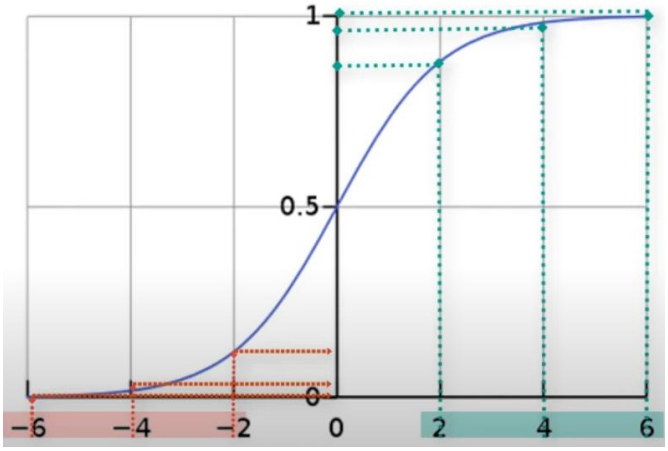
Bigger range provided by sigmoid gives more granularity in expressing other numbers.

$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

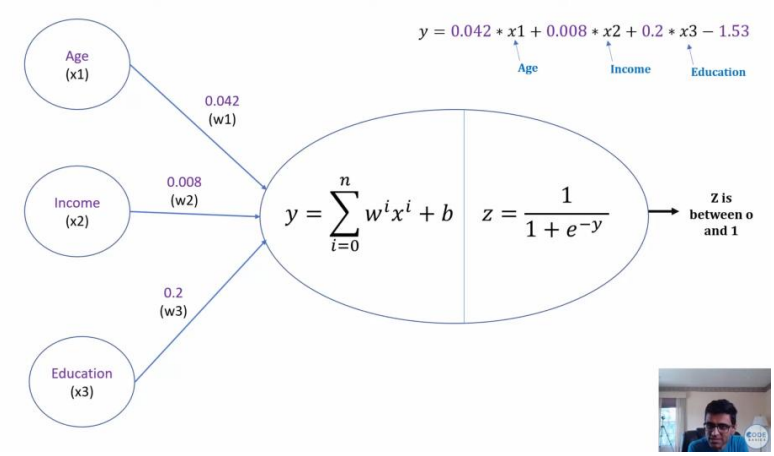
e = Euler's number ~ 2.71828

Note, although the range is -100 to 100 it is more -10 to 10 since numbers from 10 to 100 are represented by approx. 1 or -10 to -100 by approx. zero. see below that -10 is almost 0 and 10 is almost 1.

#	Sigmoid
-1000	0.0000000
-100	0.0000000
-10	0.0000454
-1	0.2689414
0	0.5000000
1	0.7310586
10	0.9999546
100	1.0000000
1000	1.0000000
1500	1.0000000

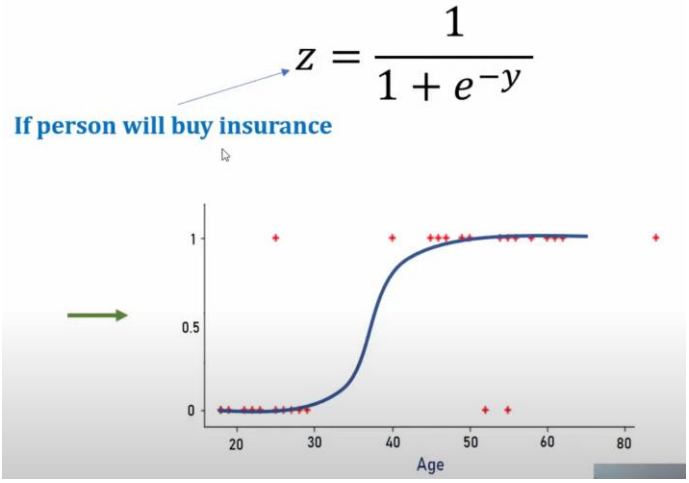


usage of sigmoid in NN:



This is where Sigmoid / Logit function comes in or Logistic Regression

- Sigmoid function splits like in graph below. The change from 0 to 1 is sharp around 35 while in Lin Reg above it is gradual and not really helping.



Lets apply sigmoid to Linear Reg numbers.

Age	Linear Reg Eqn Result Y = 0.042(age)-1.53	Sigmoid Z = 1 / (1 + e^-Y)
15	-0.9	0.2890505
20	-0.69	0.3340331
25	-0.48	0.3822521
30	-0.27	0.4329071
35	-0.06	0.4850045
36.5	0.003	0.5007500
36.429	1.8E-05	0.5000045
38	0.066	0.5164940
45	0.36	0.5890404
50	0.57	0.6387632
68	1.326	0.7901782
72	1.494	0.8166779
79	1.788	0.8566819

- Sigmoid further converted the range of numbers to 0 to 1.
- Sigmoid alone on the input does not help.

age	sigmoid
15	0.9999997
20	1.0000000
25	1.0000000
30	1.0000000
35	1.0000000
36.5	1.0000000
36.429	1.0000000
50	1.0000000
55	1.0000000

- Sigmoid distribution is – 100 to 100 so unscaled numbers don't give useful results as above:

#	Sigmoid
-1000	0.0000000
-100	0.0000000
-10	0.0000454
-1	0.2689414
0	0.5000000
1	0.7310586
10	0.9999546
100	1.0000000
1000	1.0000000

- Note: To make sigmoid useful, find some way to convert numbers given to range close to -100 to 100 and then apply sigmoid to split the data.

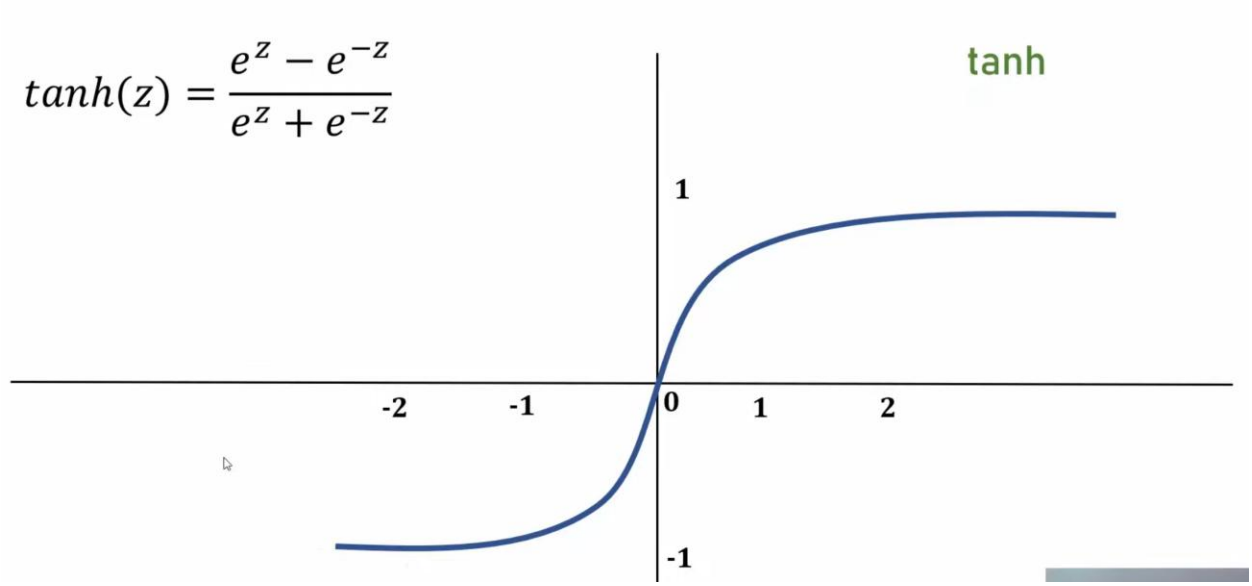
Conclusion:

The above two steps of doing scaling of the input and then putting a harsher classifier function is what a Neuron does in NN.

tanh

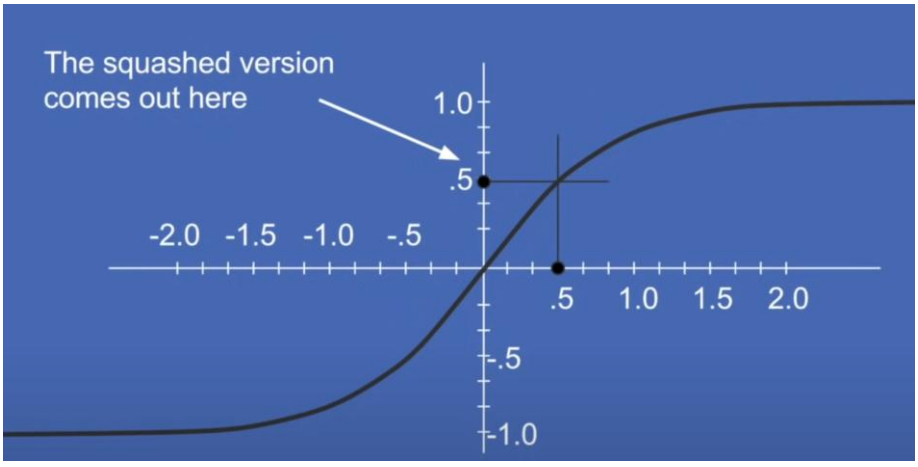
tanh converts input to number between -1 and 1.

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



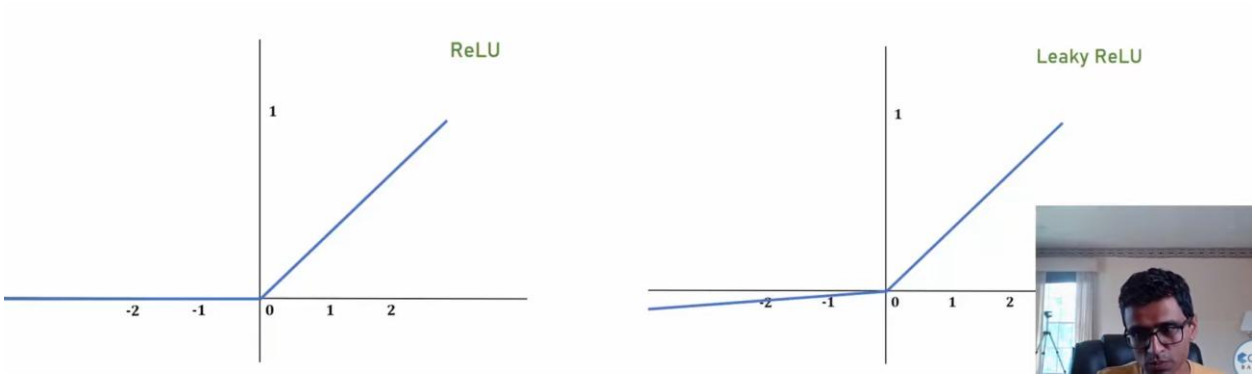
The range of real numbers it represents is -100 to 100 but just like sigmoid it is really -10 to 10 where numbers beyond 10 are almost 1.

#	Sigmoid	tanh
-1000	0	-1
-100	0	-1
-15	3.05902E-07	-1
-10	0.0000454	-0.999999996
-1	0.2689414	-0.761594156
-0.5	0.377540669	-0.462117157
0	0.5	0
0.5	0.622459331	0.462117157
1	0.7310586	0.761594156
10	0.9999546	0.999999996
15	0.999999694	1
100	1	1
1000	1	1



ReLU

ReLU: max(0,x) - - No negative, anything positive is as is



#	ReLU	leaky ReLU	Sigmoid
-1000	0	-100	0
-100	0	-10	0
-15	0	-1.5	3.05902E-07
-10	0	-1	0.0000454
-1	0	-0.1	0.2689414
-0.5	0	-0.05	0.377540669
0	0	0	0.5
0.5	0.5	0.5	0.622459331
1	1	1	0.7310586
10	10	10	0.9999546
15	15	15	0.999999694
100	100	100	1
1000	1000	1000	1

leaky ReLU: max(0.1*x,x)

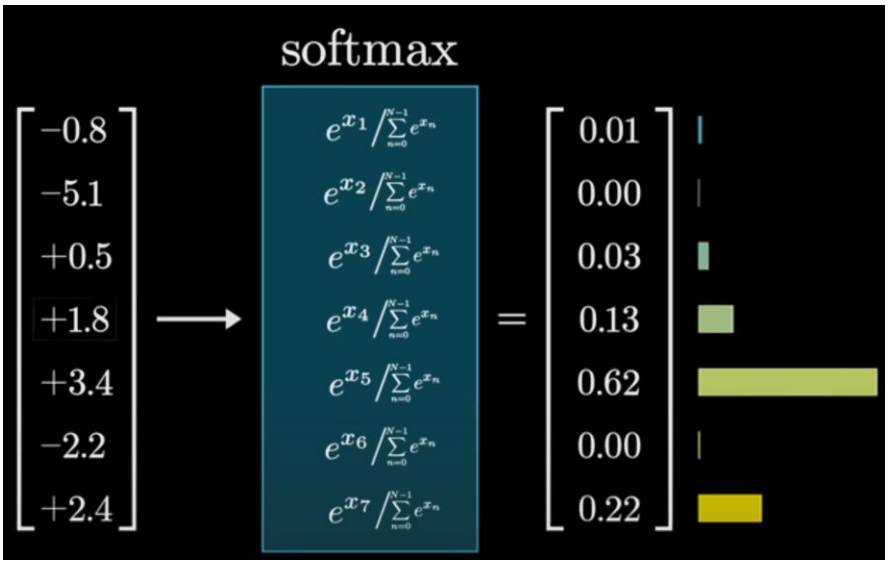
Output Layer Function:

Output layers give feedback to the final number generated through a run in the NN. Output layers have the labels like cat, dog etc. in numeric format like [1,0] as in from [cat,dog] structure. So in a real NN that guesses if a picture is cat or dog the out put will be like [0.141212, 0.85123] where these numbers are like probabilities for each class cat,dog. More on that below and why softmax is usually used in output layers.

Softmax

1. Raises all values given to the power of e
   - $e^{-0.8}$, $e^{-5.1}$, $e^{0.5}$ ... : These e^x terms are called **Logits**
2. Sums all the e^x values created : $e^{-0.8} + e^{-5.1} + e^{0.5}$ ... and divides each with the sum

The output value created is like a probability distribution with a total of 1.



$$\sigma(x) = \frac{e^x}{\sum_{i=1}^{k} e^x}$$

Softmax

- Difficult to plot because it returns a probability distribution across classes
  - 3 dimension limit
- Returns vector of probability (k) given input (x)
- argmax() = element with highest value in vector
  - Most probable class

- Output of Softmax: The output of a Softmax is a **vector** (say v) with probabilities of each possible outcome. The probabilities in vector v sums to one for all possible outcomes or classes.
- Eg: If input is (-10,8,10,20), then Output could be (0.002,0.2,0.3,0.5) thus if used in say classification based on the output 4th class is most probable class
- Argmax() returns the index position of the highest value thus the predicted class and not a probability number like 0.5 as above line.
- Negative values get assigned a super small positive number

The softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if an input is large, then it turns it into a large probability, but it will always remain between 0 and 1.

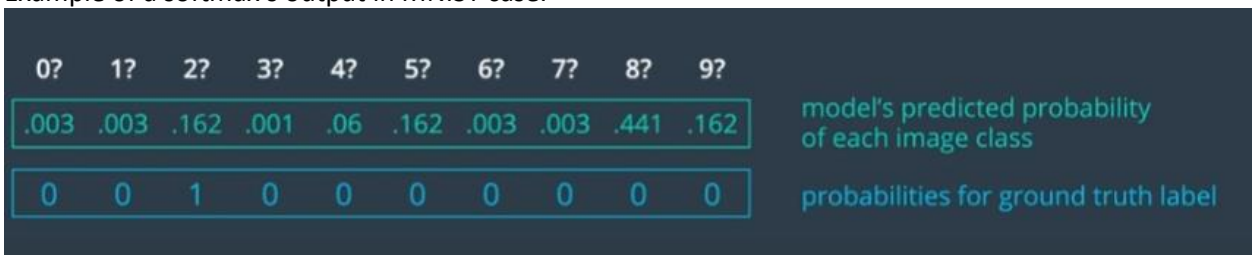
The softmax function is sometimes called the softargmax function, or multi-class logistic regression. This is because the softmax is a generalization of logistic regression that can be used for multi-class classification, and its formula is very similar to the sigmoid function which is used for logistic regression. The softmax function can be used in a classifier only when the classes are mutually exclusive.¶

Many multi-layer neural networks end in a penultimate layer which outputs real-valued scores that are not conveniently scaled and which may be difficult to work with. Here the softmax is very useful because it converts the scores to a normalized probability distribution, which can be displayed to a user or used as input to other systems. For this reason it is usual to append a softmax function as the final layer of the neural network.

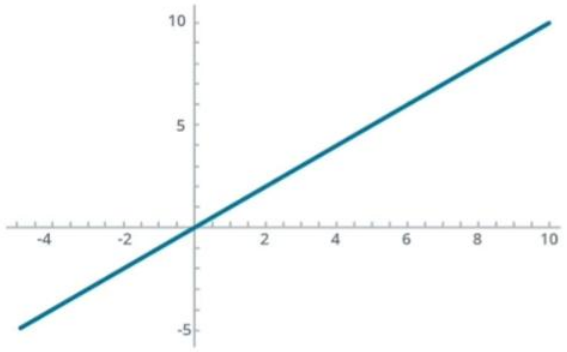
Eg2:

Formula	Input	Output	
$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$	$\begin{bmatrix} 8 \\ 5 \\ 0 \end{bmatrix}$	$\sigma(\vec{z})_1 = \frac{2981.0}{3130.4} = 0.9523$ $\sigma(\vec{z})_2 = \frac{148.4}{3130.4} = 0.0474$ $\sigma(\vec{z})_3 = \frac{1.0}{3130.4} = 0.0003$	Thus any set of numbers thrown at softmax, gets transformed to a number 0 to 1 while sum of all outputs is also 1 thus probabilities
		Numerator for each	denominator
		$e^{z_1} = e^8 = 2981.0$ $e^{z_2} = e^5 = 148.4$ $e^{z_3} = e^0 = 1.0$	$\sum_{j=1}^{K} e^{z_j} = e^{z_1} + e^{z_2} + e^{z_3} =$ $2981.0 + 148.4 + 1.0 = 3130.4$

Example of a softmax's output in MNIST case.



Identity Function:



Identity

- Predict a value from a continuous range
- Bound output from above or below using max() or min()

- Almost similar to ReLU:
  For positive numbers, the identity and ReLU are the same, but while the identity function just returns its input, ReLU needs to check if the input is greater than zero.

- Bounding the value like max(0,output) which is same as ReLU but of course here we can customize more to specific requirement

How to Choose an Output Function
Your choice of output function depends on two primary factors about what you are trying to predict: its **shape** and its **range**.

This means that our output function is determined by what we're trying to do : *classification* or *regression*. Common output functions include:

- **Sigmoid for binary classification**
- **Softmax for multi-class classification**
- **Identity or ReLU for regression**
    - There are some cases in regression where returning negative values does not make sense, so ReLU makes for a good output function than Identity.

Choosing Activation Function:

Layer	Function	Reasons and Use cases
Hidden		
Hidden		
Hidden		
Output	Relu	• Output is 0 to max while negative is pointless Eg: Speed on the road. Cars are zero speed or higher Relu does not have a max bound so setting one helps else it might guess a super high speed •
Output	Sigmoid	
Output	Softmax	• Road sign classification use case because not all road signs demand immediate stop majority are pure info so Softmax helps •

Overfitting and Underfitting:

Overfitting is when models gets overtrained ("like mugging in real life") towards the training data and is too good in predicting on training data but when shown some real world data it shows very high errors.

Underfitting is when model is too simplistic and havent captured all the nuances of the task. Get lots of errors on data it was trained on itself.

When is it overfit and when underfit: underfit shows during training itself with high errors with data from same sample population (train/test split) while overfit show low errors results during train/test/validation but high errors during real world test. Overfitting gets more attention than underfitting.

Simplest way to mitigate overfitting:

- Get more training data; diverse data
- reduce features
- Simplify the model like reduce layers/neurons in a layer in a NN etc
- Increase Dropout % (ignore a % of neurons in a layer)
- Data augmentation: modify the training data a little here and there to skew (eg: rotating pictures, zooming in or out of pics in training set) training data so model learns to differentiate these augmented data as well
- Regularization as below

Simplest way to mitigate underfitting:

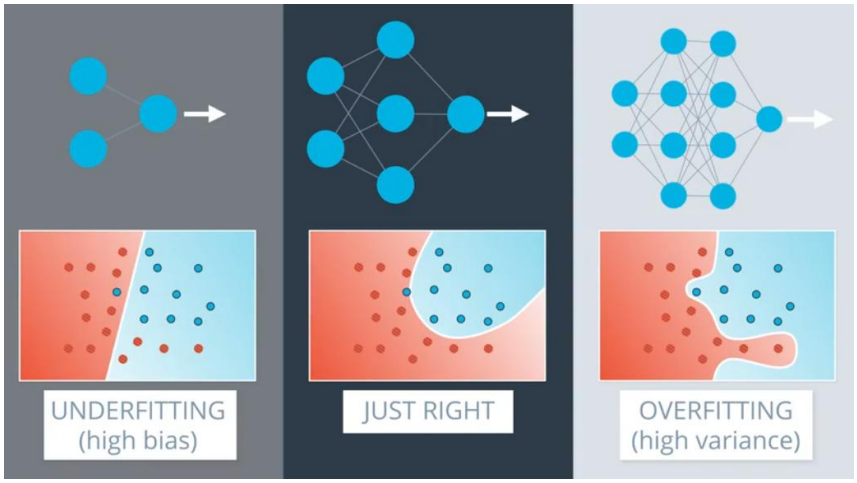
-
- Increase features
- Complexify the model with more layers/neurons in a layer in a NN etc

- Decrease Dropout %
- Data augmentation: modify the training data a little here and there to skew (eg: rotating pictures, zooming in or out of pics in training set) training data so model learns to differentiate these augmented data as well
- Regularization as below

NN Regularization

Too Many or Too Large Coeff → Overfitting

Solution: Regularization → add a function of coeffs to Error so if too many or too large coeffs happen, the error is high and those nodes/weights get punished



Regularization is counter for overfitting and high dimensionality (too many input variables).
It is done by adding a fraction of all the coefficients to the error/loss function, such that variables with small coeffs are zeroed or made insignificant.

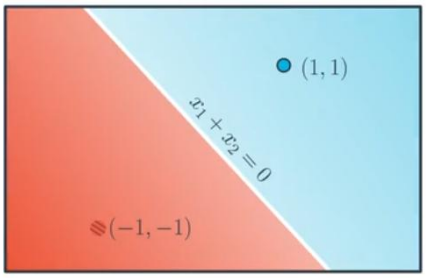
L1(Lasso) Regularization in ML

Lasso is done by taking the absolute value of weights and multiple with lambda. Due to the absolute value and no squared value, the penalty effect can lead to a zero thus nullifying a variable. This is more useful in non-neural networl ML where low contributing variables pushed to zero when the Lasso regularization is applied.

- Hence very helpful on feature engineering

Disadvantage of large coefficients:

In above figure, the two models produce the same Linear Regression Line but the one on right has a scalar multiplication or larger coefficients.
Now when the gradients are calculated, The right model gives answers which are too certain/overfit



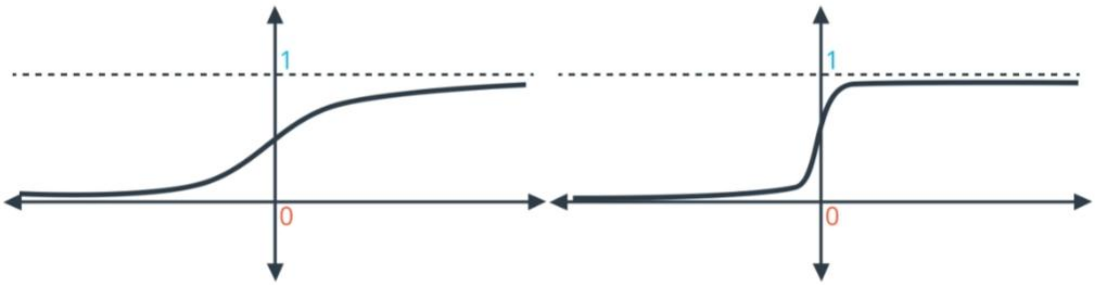
QUIZ: WHICH GIVES A SMALLER ERROR?

Prediction: $\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$

- SOLUTION 1: $x_1 + x_2$
  Predictions:
  $\sigma(1 + 1) = 0.88$
  $\sigma(-1 - 1) = 0.12$
- SOLUTION 2: $10x_1 + 10x_2$
  Predictions:
  $\sigma(10 + 10) = 0.9999999979$
  $\sigma(-10 - 10) = 0.0000000021$

What above does is that the gradients become very difficult to calculate with a smaller window as seen in the right chart below.

ACTIVATION FUNCTIONS



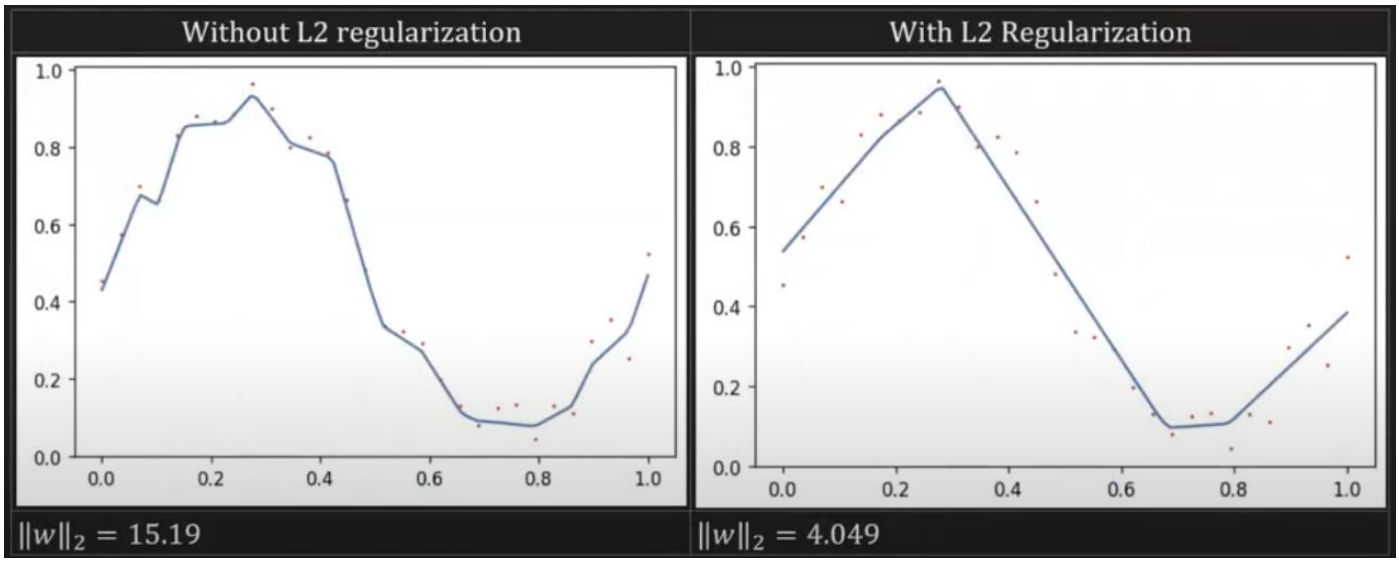
$\sigma(x_1 + x_2)$ Prediction: $\hat{y} = \sigma(x_1 + x_2 + b)$ $\sigma(10x_1 + 10x_2)$

LARGE COEFFICIENTS ⟶ OVERFITTING

In the right graph, the result will be either close to 0 or close to 1 with little room in between which is what makes the model too certain for its own good. Thus large coeffs are not good for the model and small coeffs are good. L2 Regularization forces this.

L2(Ridge) Regularization / Weight Decay in NN

Intuition: NN tries to basically make very curvy and complex line to fit all the points as close as possible all to reduce error. But this level of flexibility is not needed.
L2 regularization basically smoothes the line a bit like how smoothing / running averages smoothes time series data.



The way this is done is that the weights are penalized and a controlling factor “lambda” is used. Larger the lambda larger is the regularization. L2 Ridge regularization is a squared value like w^2 and due to this the values it effects gets close to zero but never zero.

- L2 Regularization is aka weight decay in neural networks domain and in pytorch lambda is set with a hyperparam called weight_decay

L1	L2
Cutoff less significant coeffs to zero	Keep all coeffs small

L1 Regularization

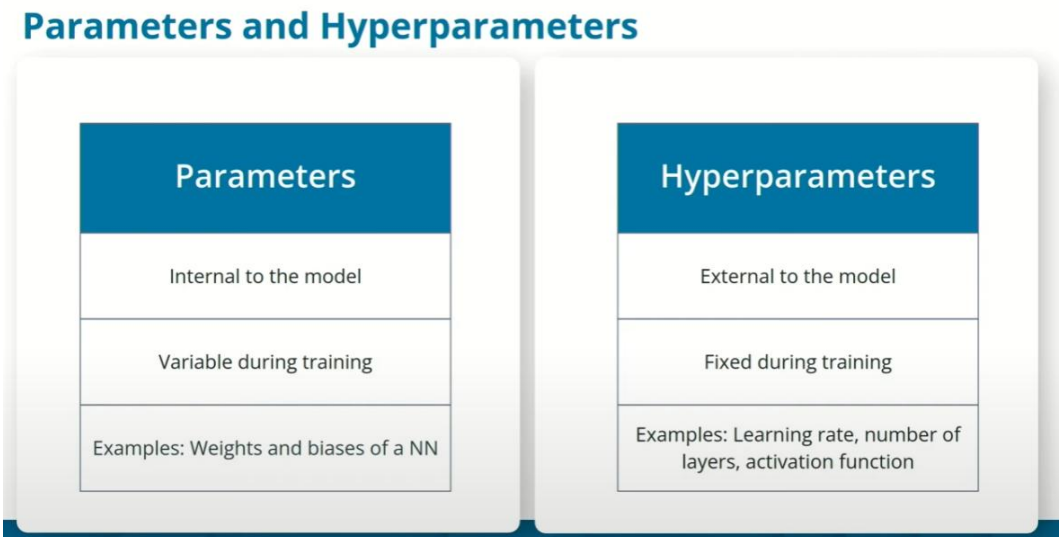
- L1 tends to result in sparse vectors. That means small weights will tend to go to zero.
- If we want to reduce the number of weights and end up with a small set, we can use L1.
- L1 is also good for feature selection. Sometimes we have a problem with hundreds of features, and L1 regularization will help us select which ones are important, turning the rest into zeroes.

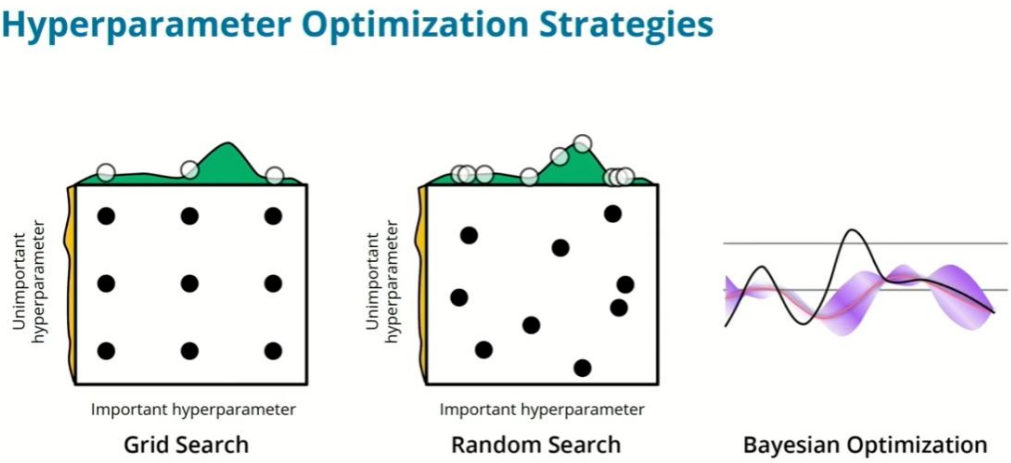
L2 Regularization

- L2 tends not to favor sparse vectors since it tries to maintain all the weights homogeneously small.

- L2 gives better results for training models so it's the one we'll use the most.

Hyperparameters:





Note: In pic above in GridSearch because few hyparams are kept constant while others experimented, we get lesser data. See how the three uniform columns end up experimenting only 3 spots despite doing 9 experiments. This is where RandomSearch comes in and randomly changes all the hyparams in each experiment leading to more useful data back.

https://deeplizard.com/learn/video/U4WB9p6ODjM#:~:text=Put%20simply%2C%20the%20batch%20size,to%20the%20network%20at%20once.

Hyperparameters / Parameters vs Powers of 2?

If you notice in the wild parameters or hyperparameters are in the powers of 2 : 2x: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 etc.
They are so due to **better memory efficiency, computational speed, and hardware alignment** in deep learning models **although not a strict requirement**. Why?

1. GPU Memory & Computational Efficiency

- GPUs are optimized for matrix operations, and **powers of 2 align with memory access patterns**.
- **Matrix multiplication (dot product of weights and inputs)** runs faster when the dimensions are powers of 2, reducing computational overhead.

2. Hardware-Level Parallelism

- Modern GPUs, including your **Tesla T4**, use **SIMD (Single Instruction, Multiple Data)** and **Tensor Cores** that process data in chunks (often multiples of 16 or 32). Using **powers of 2** ensures better utilization of these parallel units, preventing **wasted computational cycles**.

3. Memory Alignment & Avoiding Padding

- In deep learning, tensors are stored in memory as contiguous blocks. If the number of neurons is **not a power of 2**, extra padding might be needed, which increases memory usage **without improving performance**.

4. More Stable Training

- Network architectures with **powers of 2 neurons/filters** tend to have **more stable gradient propagation**.
- This can help avoid vanishing/exploding gradients in deep networks.

Can You Use a Batch Size of non power of 2 like 80?

✅ Yes, you can use a batch size in CNN of 80—nothing will break! ❌ But **it might be slightly slower**

Should You Always Use Powers of 2?

✅ **Recommended** for:

- **Neurons** in fully connected layers (e.g., 128, 256, 512)
- **Feature maps in CNNs** (e.g., 32, 64, 128, 256)
- **Batch sizes** (e.g., 16, 32, 64, 128)

Other Hyperparams:

1. Learning Rate Decay Steps in LR Scheduler
2. Hidden Layer Sizes in RNNs / LSTMs / Transformers
3. Number of Attention Heads in Transformers
4. Projection Dimensions in Word Embeddings
   Why? In word embeddings, dimensions are often 256, 512, 1024, ensuring alignment with tensor operations.
5. CNN
   1. **Pooling Window Sizes** (in CNNs & Downsampling)
   2. Number of Filters in CNN Layers

❌ **Not strictly necessary** for:

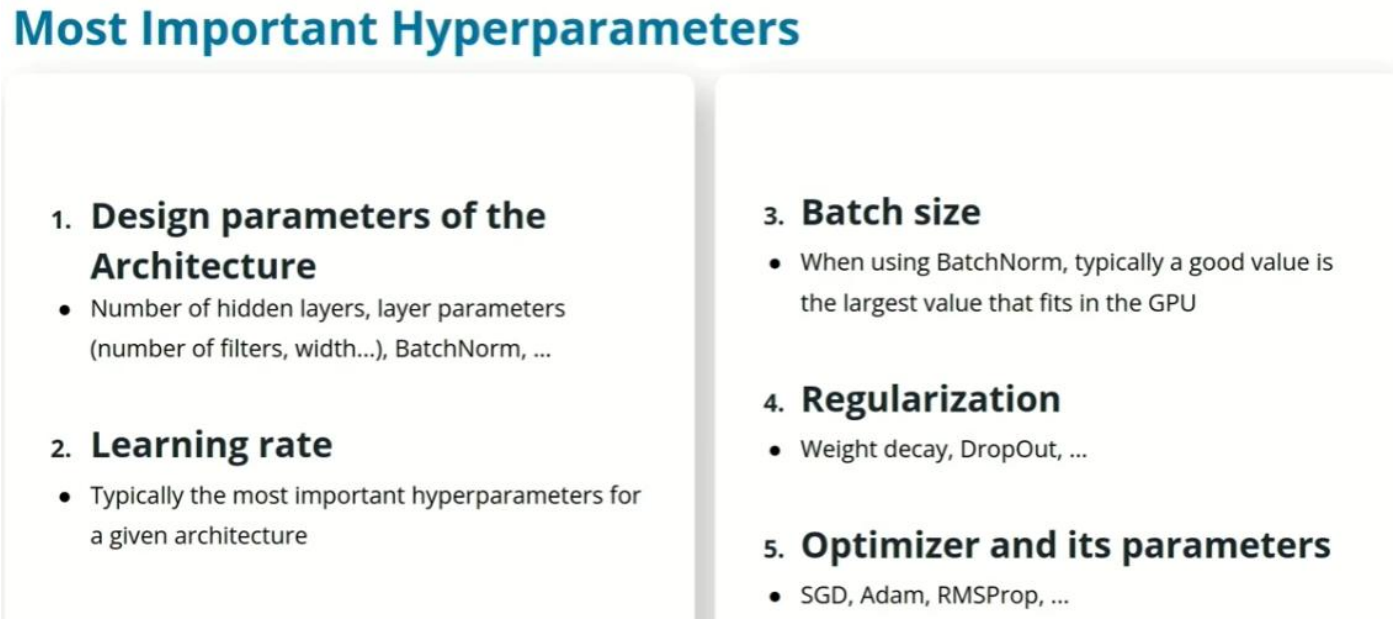
- **Dataset size** (since you can randomly sample mini-batches)
- **Custom architectures** (sometimes odd numbers work better based on experiments)

Strategies for Optimizing Hyperparameters

- **Grid search**
  - Divide the parameter space in a regular grid
  - Execute one experiment for each point in the grid
  - Simple, but wasteful
- **Random search**
  - Divide the parameter space in a random grid
  - Execute one experiment for each point in the grid
  - Much more efficient sampling of the hyperparameter space with respect to grid search
- **Bayesian Optimization**
  - Algorithm for searching the hyperparameter space using a Gaussian Process model
  - Efficiently samples the hyperparameter space using minimal experiments

Most Important Hyperparameters

Ranked Order!



Optimizing hyperparameters can be confusing at the beginning, so we provide you with some rules of thumb about the actions that typically matter the most. They are described in order of importance below. These are not strict rules, but should help you get started:

1. Design parameters: When you are designing an architecture from scratch, the number of hidden layers, as well as the layers parameters (number of filters, width and so on) are going to be important.
2. Learning rate: Once the architecture is fixed, this is typically the most important parameter to optimize.
3. Batch size: This is typically the most influential hyperparameter after the learning rate.
4. Regularization
5. Optimizers

Most Common Hyperparameters:

Nature of Target variable, Expected Output value nature, Output Activation Function, Error Function type

Binary Classification	Multiclass Classification	Regression
Target is 0 or 1	Vector of targets {0, 1}	Target is a numerical value
Output probability of label	Output probability vector	Output predicted value
Sigmoid activation	Softmax activation	Identity activation
Binary Cross Entropy	Multi-class Cross Entropy	Mean Squared Error

Strategies for Hyperparameter Tuning:

Hparam	What goes up	What goes down
Neurons	Increasing the neurons in each layer increases **the model's capacity** to learn complex features	$, model complexity, more time
Filters in CNN	Increasing the number of filters helps capture **more diverse** and **detailed** features as we go deeper.	
Activation	In deep networks, **ReLU neurons can "die"** (outputting zero for all inputs). This happens when gradients become **too small** during training. **Leaky ReLU instead of ReLU allows a small gradient (α ≈ 0.01) for negative inputs**, preventing dead neurons. **Leaky ReLU provides better gradient flow**, which might **help the network extract finer details** from images. Small datasets (like your **5000 images**) are more prone to poor gradient flow issues.	
Batch Size		

TroubleShooting with Hyperparameter Tuning

Symptom	Possible Hparam	What could be happening?	How to Confirm	What to do?
NaN or Inf Values in Loss OR Sudden Spikes in Loss within batches in same epoch OR Models preds seem random	Activation	Gradient Explosion	1) To confirm, PyTorch has a built-in tool to catch NaN/Inf gradients called detect_anomaly() To confirm another way, we can monitor gradient's norm/magnitude. If gradients explode, their L2 norm (magnitude) will increase drastically	1)Gradient Clipping – Set a max gradient norm 2) Lower LR 3) Use BatchNorm 4) Normalize Data Input

OR Loss stops decreasing or jumps to Inf/NaN				

Myths in Optimizations that might not work:

- Sorry, while **tracking your experiments** is definitely a good idea, it does not per se improve your performance (although it can help you to find the right experimental direction).
- Sorry, **using larger networks** does not necessarily increase performance. It can even decrease performance if you start overfitting.
- Sorry, **training for longer** does not necessarily increase your performance. It might even decrease your performance if you overtrain (it leads to overfitting).

Vanishing and Exploding Gradient

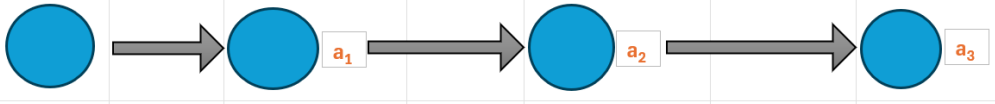
During back propag, the gradient (number 0 to 1) by formulaic application, gets smaller and smaller due to multiplication of fractions (0<n<1) and thus in deep networks the gradient becomes too small before it reaches the start.

Activation func like sigmoid that are bounded leads to Vanishing Gradient during Back Propag.

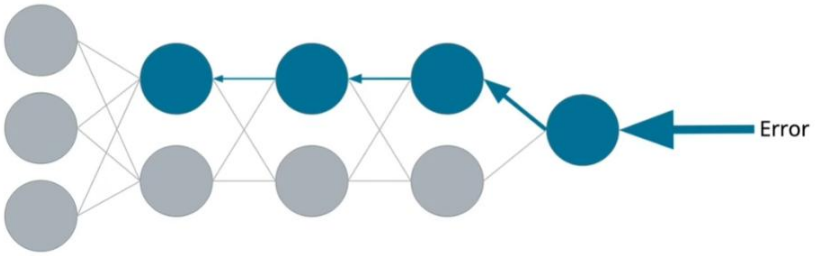
Below is an example of how the gradient vanishes.
Situation:

- Forward Pass: Input = 1, Assume all weights = 0.5
- Activation Function = Sigmoid

- Back Propag: Error = 0.1, LR = 0.01
  - Partial Derivative of Activation Func, Sigmoid = Ax * (1 – Ax) = Activated Value * (1 - Activated Value)
  - Gradient = PD * BPInput → (Partial Derivative * Appropriate input coming thro Back Propag)
  - BPInput is the Error when Back Propag is started and after that, the Gradient * Weight from previous step becomes the BPInput
  - Note that BPInput already is small and when LR is applied it becomes even small and so the update to the weight is super small

					G2 is dependent on G3 * w3 from previous layer	With LR and small Gx, the udpate to weight is super small		**Back Propag:**
			PD1 = 0.235		PD2 = 0.244		$PD_3 = a_3 * (1 - a_3)$ = 0.5717 * 0.1 = 0.244	Formulas: **Gradient, Gx = PDx * $Input_{BP}$.** $Input_{BP}$ is E at the last node and Previous Gradient for earlier nodes
			G1 = PD1 * w2 * G2 = 0.235 * 0.5 * 0.002 = 0.0003		G2 = PD2 * **w3** * **G3** = 0.244 * 0.5 * 0.024 = 0.002		G3 = PD3 * E = 0.244 * 0.1 = 0.02	**Partial Derivative, PDx = a_x * (1 - a_x)** As applicable to Sigmoid formula. Diff Act. Func will have diff PD formula
			G1 = 0.0003		**G2 = 0.002**		**G3 = 0.02**	E = 0.1 , LR = 0.01
		$w1_{new}$ = 0.499997		$w2_{new}$ = 0.49997		$w3_{new}$ = 0.4997		$w_{new} = w_{old}$ - LR * Gx
								
	Input		Hidden 1		Hidden 2		Output	
Feed Forward:		**w1 = 0.5**		**w2 = 0.5**		**w3 = 0.5**		
a_x = Neuron's final output at a layer. value after activation func is applied on incoming values	Input = 1		$a_1 = \sigma (0.5 * 1)$		$a_2 = \sigma (0.5 * 0.622)$		$a_3 = \sigma (0.5 * 0.577)$	
Act. Func = Sigmoid **$a_x = \sigma (w_x * Input_{FF})$**			a_1 = 0.622		a_2 = 0.577		a_3 = 0.5717 **(Final Output)**	
$Input_{FF}$ is prev. layer output								

"**saturating" activation functions** -- those that have a bounded range (Sigmoid – high values get bounded to a max value of 1), can cause vanishing gradients. Meanwhile, unbounded activation functions (ReLU – in which a very high value is used as is) can cause exploding gradients.

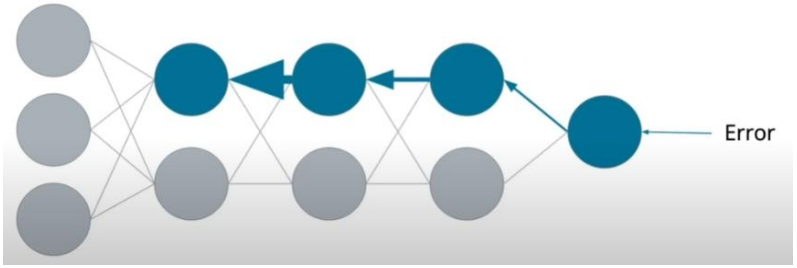


- Counter is to use Non-saturating function like ReLU which are un-bounded
- Or use Adam (Adaptive Moment Estimation) optimizer instead of Gradient Descent
    - LR does not vanish
    - Converges quickly

Exploding Gradient is caused when weights greater than 1 multiply against each other during back propag and become too large. The rate of increase/explosion is exponential scale.

Non saturating func's gradients are greater than 1 and so the gradient keeps increasing during back propab

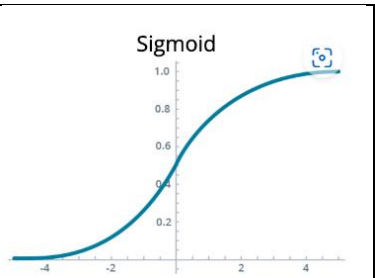
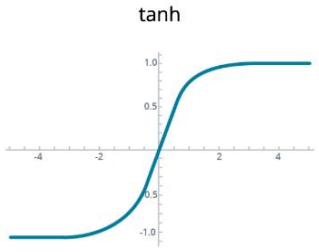
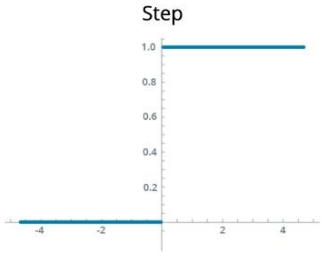
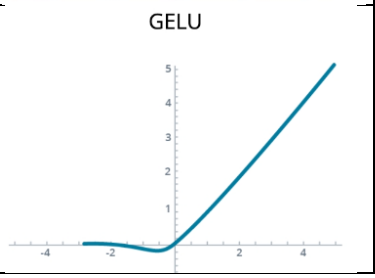
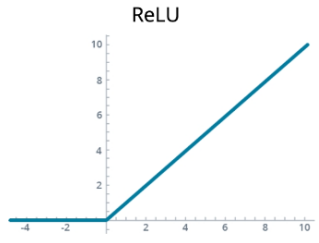
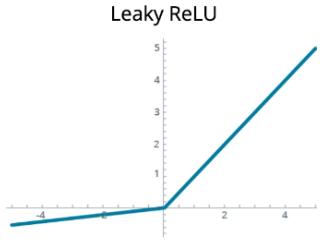
- Counter for Exploding = **Gradient Clipping & Weight Regularization**
    - we can use the torch.nn.utils.clip_grad_norm_ (PyTorch) function to clip our gradients by bounding the norm of the gradient (does not bound the value of the gradient but norm).
- We can also use the clamp (PyTorch) method to restrict the minimum and maximum size of our tensors.



You monitor below in tool like TensorBoard where each neuron weights and loss are tracked:

Vanishing Gradients	Exploding Gradients
Neuron weight drops	Large neuron weight
High, stalled training loss	Unstable loss

- Unstable = drastic changes in gradients and even going to NaN

Saturating Func			
Non-Saturating func			

Weight Initialization

Before the introduction of BatchNorm, weight initialization was really key to obtaining robust performances. In this previous era, a good weight initialization strategy could make the difference between an outstanding model and one that could not train at all.

However, a good weight initialization can speed up your training and also give you a bit of additional performance. In general, weights are initialized with random numbers close but not equal to zero, not too big but not too small either. This makes the gradient of the weights in the initial phases of training neither too big nor too small, which promotes fast training and good performances. Failing to initialize the weights well could result in **vanishing or exploding gradients**, and the training would slow down or stop altogether.

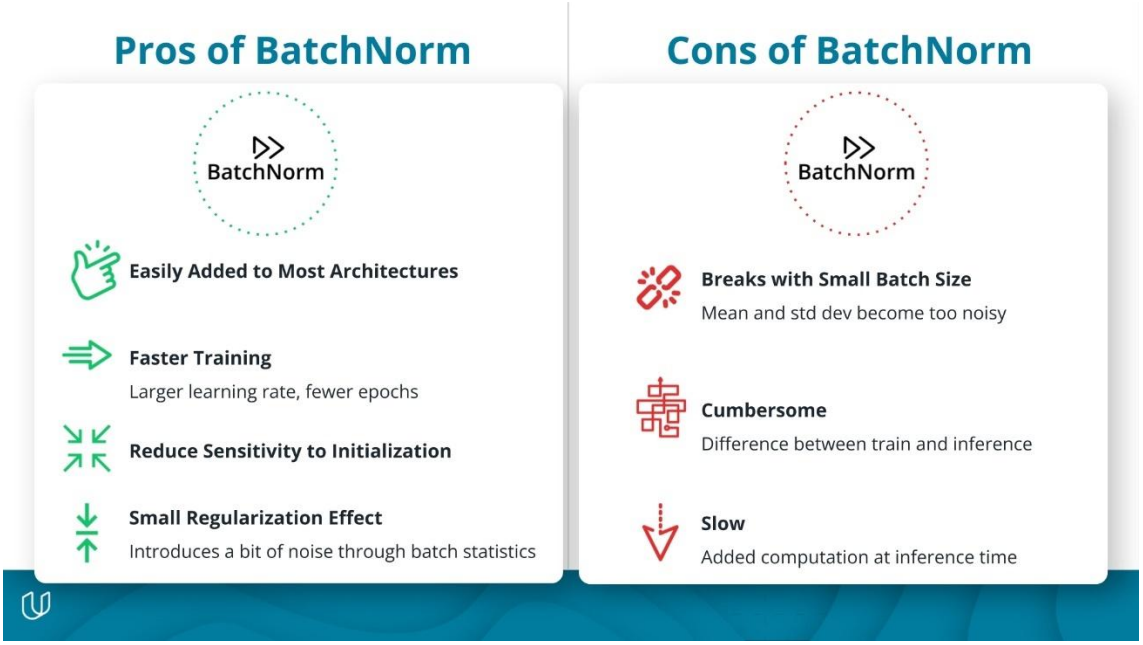
Batch Normalization

Just like how input data gets normalized before it is introduced in many ML algo, you can normalize the data between layers. Its called “Batch” since in NNs, the per-batch stats like mean and std are calculated and used instead of a global mean and std.

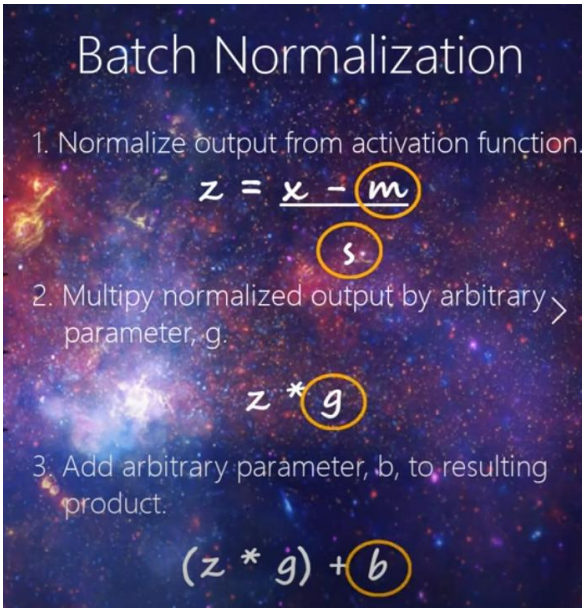
What's different in BatchNorm is that a weight and bias term are introduced which are trainable during back propag. Because of these two terms and thus additional control levers, bias term is usually removed from the layer preceding BatchNorm which will otherwise be present in most normal NN blocks. The bias in BatchNorm makes the normal bias term redundant.

- Applying BatchNorm after ReLU would normalize already transformed (non-linear) activations, which is less effective.
- Note: NN during testing phase is put on eval() mode in Pytorch. In this mode the BatchNorm layers use the global averages and not batch mean/std.

It does not usually improve the performances per se, but it allows for much easier training and a much smaller dependence on the network initialization, so in practice it makes our experimentation much easier, and allows us to more easily find the optimal solution.



What does batch norm do?

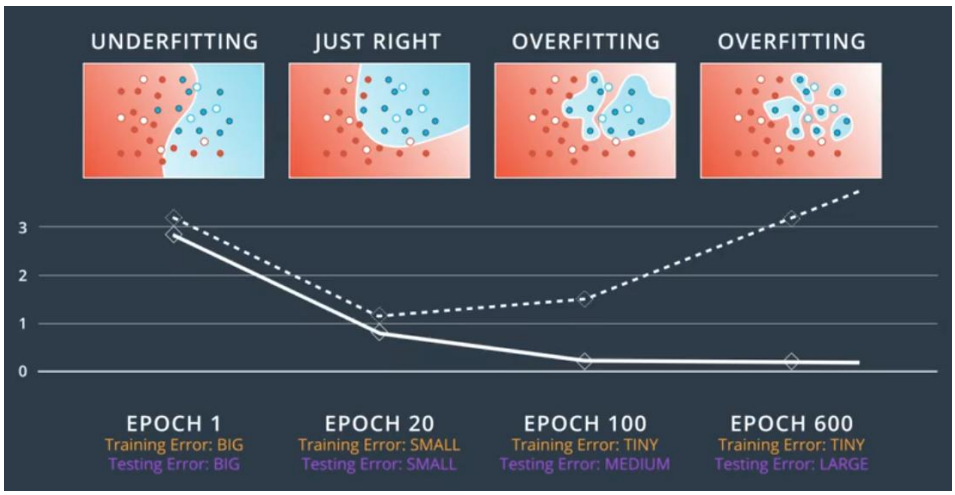


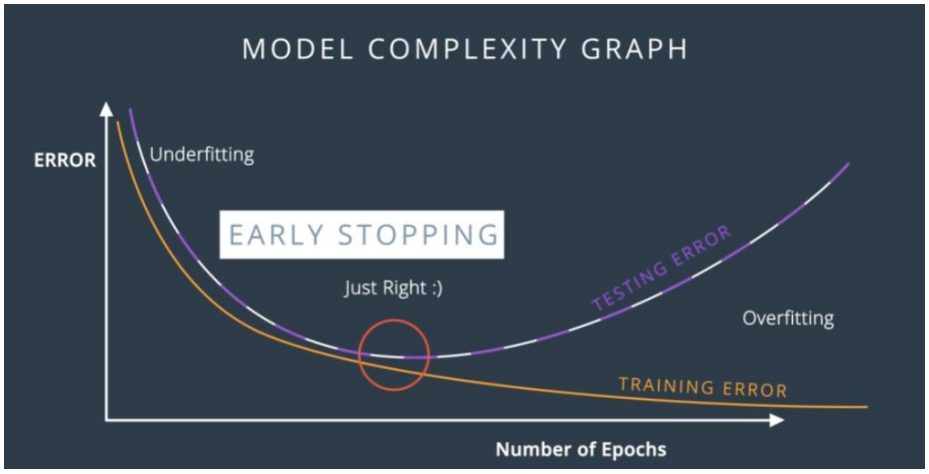
ChatGPT says BatchNorm is more effective before activation func. But some popular architectures place it after.

- New weights and bias kind of numbers applied to number after activation to “normalize”
- These numbers are also “trainable” in sense they also get changed during back propagation and affect the new weights meaningfully.

Early Stopping (How many Epochs?):

Fix Overfitting





MODEL COMPLEXITY GRAPH

Underfitting: High Bias
Overfiiting: High Variance

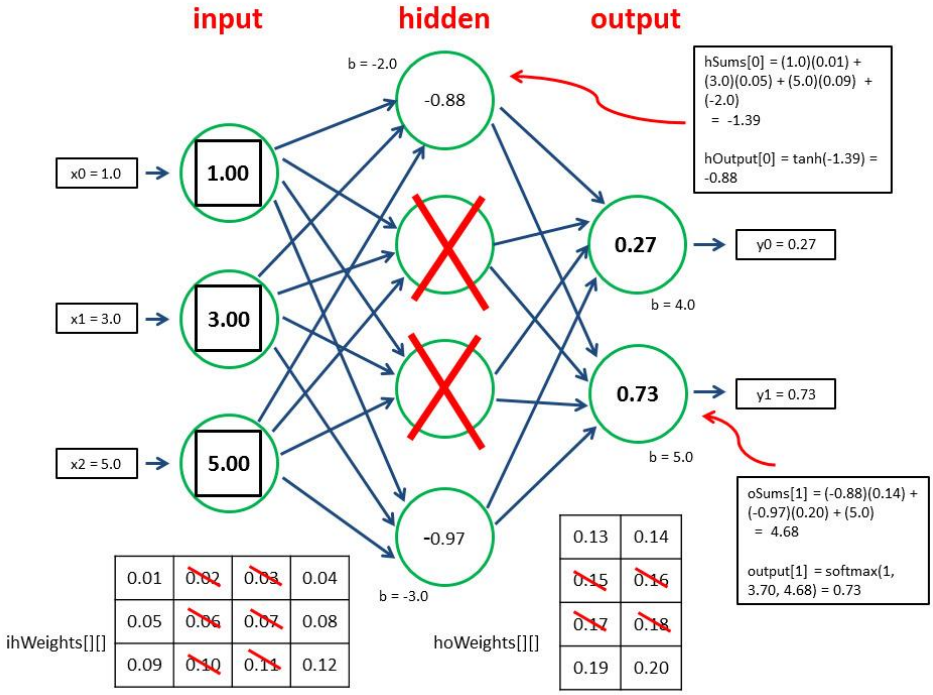
Stop when error starts to increase strategy called early stopping like elbow graph in kmeans

Dropout

When training a neural network, sometimes one part of the network has very large weights and it ends up dominating the training, while another part of the network doesn't really play much of a role (so it doesn't get trained).
To solve this, we can use a method called **dropout** in which we turn part of the network off and let the rest of the network train.

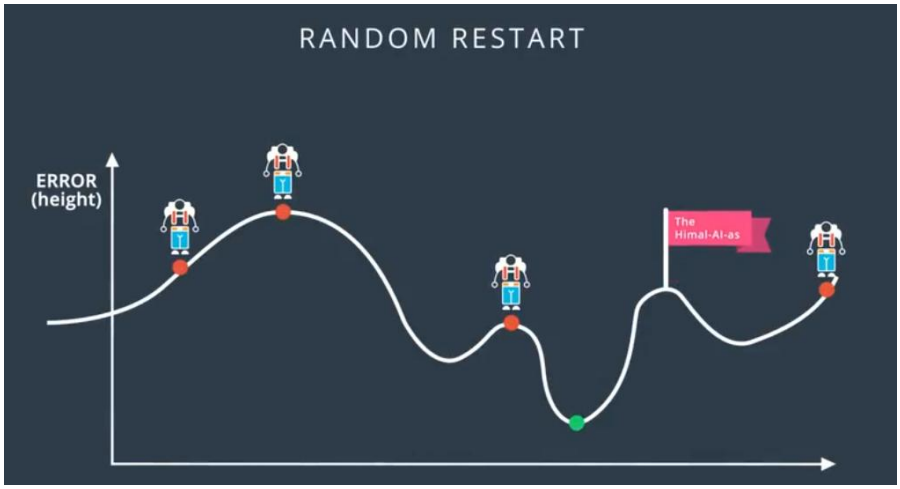
Each node is assigned a probability like say 20% and so during each Epoch/Backprob nodes get switched of 20% of the time.

- Note that some nodes may get turned off more than others and some may never get turned off. This is OK since we're doing it over and over; on average, each node will get approximately the same treatment.



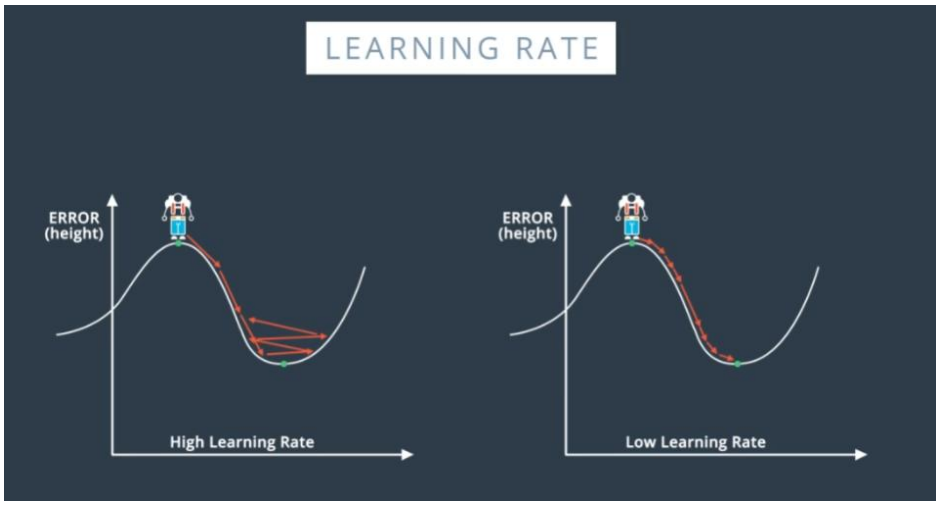
Random Restart

Gradient descent algo can get stuck in a local minima instead of the global minima. One way to solve our problem is to use random restart. We start (and restart) from a few different random places and do gradient descend from all of them. This increases the probability that we'll get to the global minimum, or at least a pretty good local minimum.



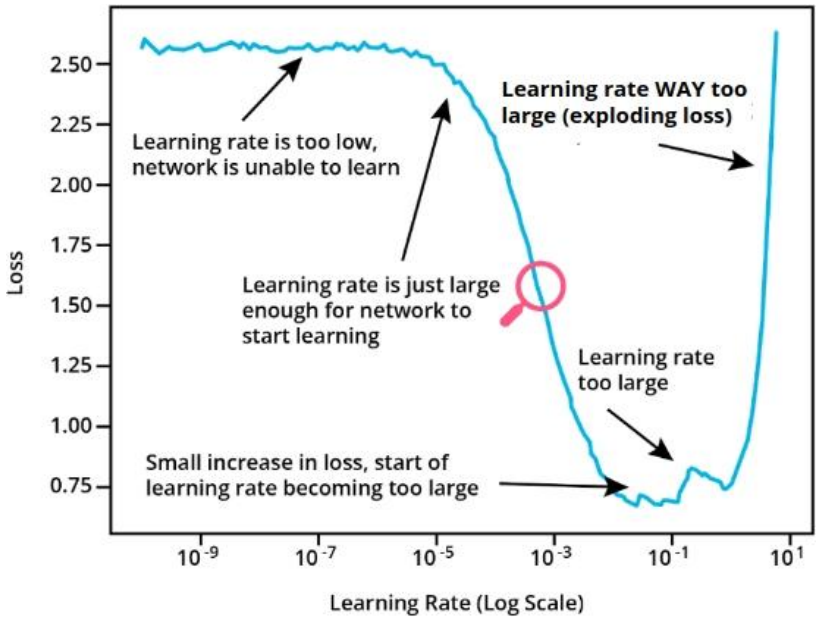
Learning Rate:

What Learning Rate to use is a research question and we don't really have answers but too big results in jumping around and not diverging while small values usually gets good results.
General rule of thumb: if model is not working reduce the LR. The best learning rates are those that decrease as the algorithm is getting closer to a solution.



Optimizing Learning Rate:

"learning rate finder." tool helps in finding optimal LR. It scans different values of the learning rate, and computes the loss obtained by doing a forward pass of a mini-batch using that learning rate. Then, we can plot the loss vs. the learning rate and obtain a plot similar to this:



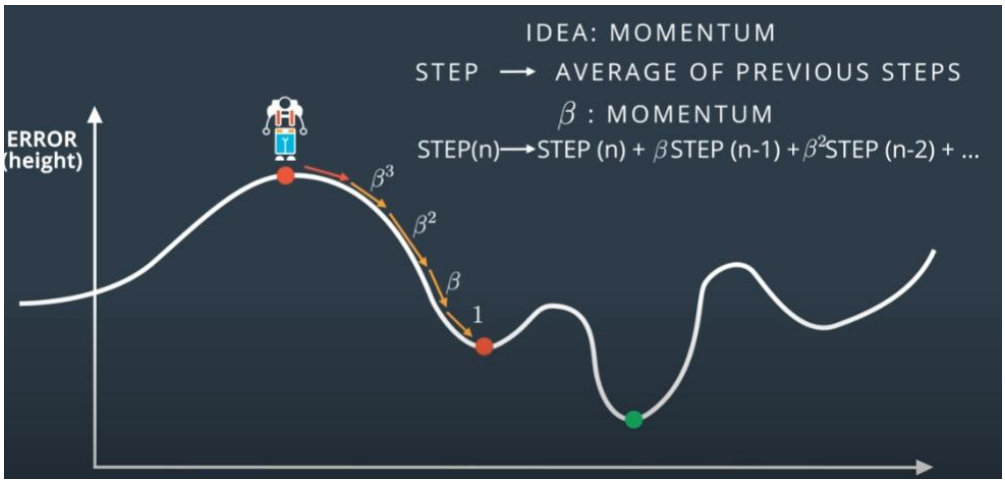
LR Finder is different from LR Scheduler as below.

PyTorch learning rate scheduler

i.e., a class that dynamically (between epochs) changes the learning rate. Basically keeps LR high in the beginning and reduces as we expect the model to be close to the minima.

There are several possible learning rate schedulers. You can find the available ones in the **documentation**.
One of the simplest one is the StepLR scheduler. It reduces the learning rate by a specific factor every n epochs.

Momentum



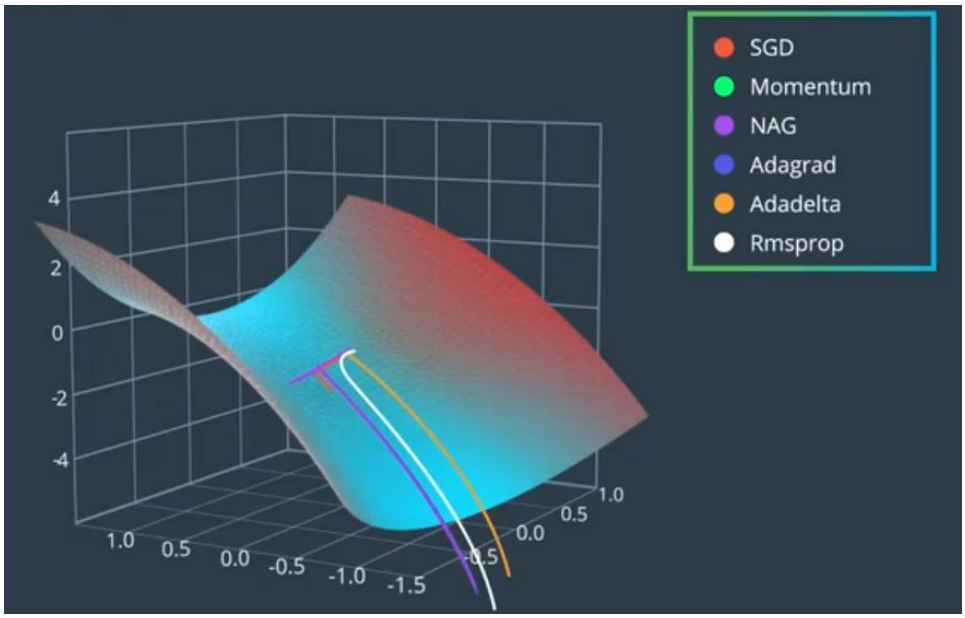
Pushing Past Local Minima

Another way to solve the local minimum problem is with **momentum**. Momentum is a constant *β* between 0 and 1. We use *β* to get a sort of weighted average of the previous steps:

$$\text{step}(n) + \beta\text{step}(n-1) + \beta^2\text{step}(n) + \beta^3\text{step}(n-2) + \dots$$

Optimizer

Adam is a popular optimizer that uses the concept of momentum to converge quickly and popular



Above from MNIST Number prediction case; SGD algo used and difference in performance of different Optimizers

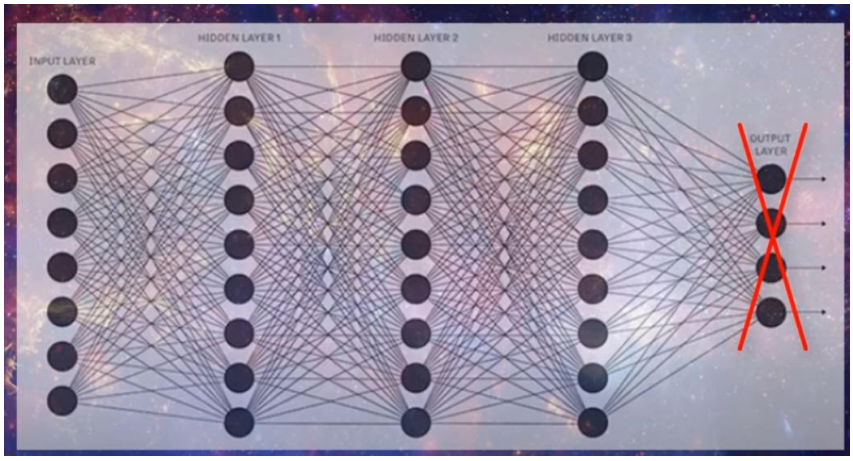
Re-purposing a NN:

https://www.youtube.com/watch?v=5T-iXNNiwIs

Sometimes there might be a NN that is trained to detect cars but not trucks. Instead of creating a new NN from scratch for trucks you can repurpose the Car detecting NN.

The idea is that the early hidden layers have all specialized in identifying something like tyres, mirrors, headlights etc

which are quite similar in trucks so lets keep the early hidden layers and get rid of the last Output layer alone. It can make sense to remove more end layers as well. Now with the preset weights from car model re-run with trucks and the model should train faster.



Design of Neural Network

Rules of Thumb
When designing NN you have a lot of different possibilities. There are no strict rules, and experimentation is key. However, here are some guidelines to help you get started:

The number of inputs input_dim is fixed (in the case of MNIST images for example it is 28 x 28 = 784), so the first layer must be a fully-connected layer (Linear in PyTorch) with input_dim as input dimension.
Also the number of outputs is fixed (it is determined by the desired outputs). For a classification problem it is the number of classes n_classes. So the output layer is a Linear layer with n_classes (in case of classification).
and for a regression problem it is 1 (or the number of continuous values to predict).

What remains to be decided is the number of hidden layers and their size. Typically you want to start from only one hidden layer. Sometimes adding a second hidden layer helps, **and in rare cases you might need to add third**. But one is a good starting point.

As for the number of neurons in the hidden layers, a decent starting point is usually the mean between the input and the output dimension. Then you can start experimenting with increasing or decreasing, and observe the performances you get.
If you see overfitting, start by adding regularization (dropout and weight decay) instead of decreasing the number of neurons, and see if that fixes it. A larger network with a bit of drop-out learns multiple ways to arrive to the right answer, so it is more robust than a smaller network without dropout. If this doesn't address the overfitting, then decrease the number of neurons.
If you see underfitting, add more neurons. You can start by approximating up to the closest power of 2. Keep in mind that the number of neurons also depends on the size of your training dataset: a larger network is more powerful but it needs more data to avoid overfitting.

Methods of Teaching NN:

Supervised Learning in NN:

Most popular application of NN and most of below notes about.
NN's are popular in Supervised learning since one of its core mechanism of back-propagation needs labeled data to quantify error and then tell which weights need reduction and which needs increase.

Unsupervised Learning in NN:

“Unsupervised” learning do exist with some cool way of its implementation.
Some cool techniques like Autoencoders (like Autobiography or self code) is a way where NN sets input image as the output image where during back propagation, the NN learns to exactly reproduce the input.

Semi-supervised Learning in NN:

Semi-supervised is basically using the small number of labeled data to label unnamed data and then fitting the NN on the OG Labeled data and NN labeled data to improve the NN model.