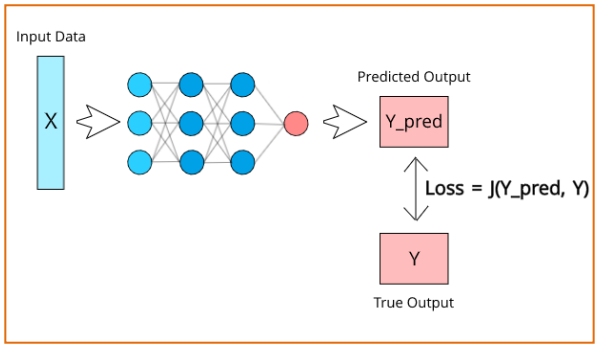
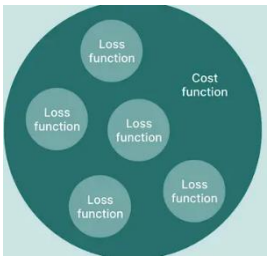
Deep Neural Network: Loss function, optimization techniques - Gradient descent, RMSprop, backpropagation, training deep models, regularization - Early stopping, augmentation, dropout.

Loss function:



- In deep learning, a loss function, also known as a **cost function or objective function**.
- It is a mathematical function that measures the difference between the predicted output and the actual output.
- Minimizing the loss function automatically causes the neural network model to make better predictions regardless of the exact characteristics of the task
- The loss function helps the model to adjust its parameters during training to minimize the error.
- **Loss Function Vs Cost Function:** A loss function/error function is for a single training example/input and a cost function, on the other hand, is the average loss over the entire training dataset.
- The choice of the loss function depends on the specific problem and the type of output being predicted.



- Uses of loss functions:
  - Loss functions are used in optimization problems with the goal of minimizing the loss.
  - Loss functions are used in regression when finding a line of best fit by minimizing the overall loss of all the points with the prediction from the line.
  - Loss functions are used while training perceptrons and neural networks by influencing how their weights are updated.
  - The larger the loss is, the larger the update.
  - By minimizing the loss, the model's accuracy is maximized.
- The following are the types of loss functions:
  - MSE(Mean Squared Error)
  - MAE(Mean Absolute Error)
  - Hubber loss
  - Binary cross-entropy
  - Categorical cross-entropy
  - KL Divergence

Mean Squared Error:

- It measures the average squared difference between the predicted values and the actual values in a dataset.
- Also called L2-loss.

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

- A lower MSE indicates that the model is better at predicting the values in the dataset.
- *Advantages*
  1. MSE penalizes the model for making large errors
  2. In the form of quadratic equation $ax^2+bx+c=0$
     - Plotting the equation will get Gradient Descent of global minimum
     - We don't get any local minima
- *Disadvantages*
  1. It is not robust to outliers, MSE penalizes the outliers most and the calculated MSE is bigger.
  2. Not suitable for imbalanced datasets

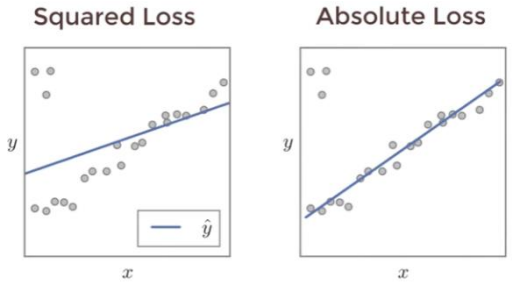
```
def MSE(y_predicted, y):
squared_error = (y_predicted - y) ** 2
sum_squared_error = np.sum(squared_error)
mse = sum_squared_error / y.size
return(mse)
```

MAE(Mean Absolute Error):

- It is calculated by taking the absolute difference between the predicted value and the actual value, and then averaging the differences over the entire set of observations.
- Also called as L1 loss.

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

- *Advantages*
  1. *MAE is less sensitive to outliers in the data.*
  2. *MAE is a linear metric, which means that changes in the errors are directly proportional to changes in the predicted values.*
  3. *MAE is a simple metric to compute*
- *Disadvantages*
  1. *It may have local minima as it has linear equation.*
  2. *Less sensitive to changes in smaller errors*
  3. *Ignores direction of errors*
  4. *The graph of MAE is not differentiable so we have to apply various optimizer like Gradient Descent which can be differentiable.*



Hubber loss

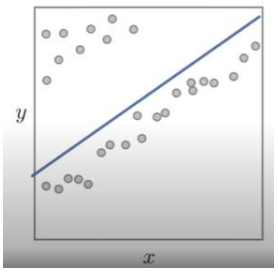
- *It is defined as a combination of Mean Absolute Error (MAE) and Mean Squared Error (MSE) loss functions.*
- *The Huber loss is similar to **MAE for small errors**, and similar to **MSE for large errors**.*

$$L_\delta = \begin{cases} \frac{1}{2}\left(y - f(x)\right)^2, & if\ |y - f(x)| \le \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & otherwise \end{cases}$$

Quadratic

Linear

- This makes it more robust to outliers compared to MSE, which is more sensitive to outliers due to the squared term.
- The use of the delta parameter allows for the trade-off between the robustness of the loss function and the sensitivity to smaller errors.
- *Advantages*
  1. *Robustness to outliers.*
  2. *Huber loss is a smooth and differentiable function, which makes it easy to use in optimization algorithms that require the computation of gradients, such as gradient descent.*
- *Disadvantages*
  1. *The choice of delta can have a significant impact on the results, and selecting the optimal value can be challenging.*
  2. *The Huber loss is more computationally expensive than the MSE loss due to its non-smoothness.*
  3. *In particular, selecting a large delta can lead to high bias, while selecting a small delta can lead to high variance.*



Note:-

- MAE and MSE is a commonly used metric in regression analysis and is often used to evaluate the performance of machine learning models.

- Huber loss is a robust, flexible, and interpretable loss function that is well-suited for regression analysis tasks involving datasets that contain outliers or noise.

Binary cross-entropy loss:

- Is a commonly used loss function for binary classification problems.
- It measures the difference between the predicted probability distribution and the actual probability distribution.
- It is generally used with logistic regression.

$$\text{Loss} = -\frac{1}{\underset{\text{size}}{\text{output}}} \sum_{i=1}^{\underset{\text{size}}{\text{output}}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

- *Advantages*
  1. *It is a smooth, continuous, and differentiable function that is well-suited for optimization algorithms that require gradients.*
  2. *It penalizes the model more heavily for making incorrect predictions.*
  3. *It provides a probabilistic interpretation of the model's predictions, which can be useful for decision-making.*
- *Disadvantages*
  1. *It can be sensitive to outliers and noise in the data, which can lead to overfitting.*
  2. *It assumes that the data is binary and that the probabilities are independent of each other, which may not be true in all cases.*

Categorical cross-entropy loss:

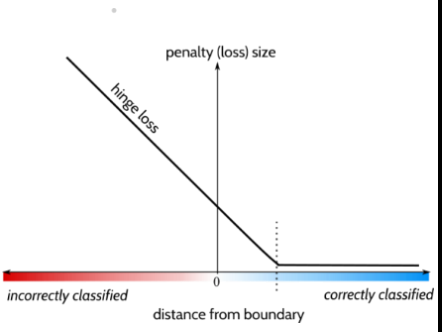
- Is a commonly used loss function for multi-class classification problems.
- It measures the difference between the predicted probability distribution of the output and the actual probability distribution of the output.

$$\text{Loss} = -\sum_{i=1}^{\underset{\text{size}}{\text{output}}} y_i \cdot \log \hat{y}_i$$

- *Advantages*
  1. *A differentiable function, which means it can be used with gradient-based optimization.*
  2. *Encourages model to output high probabilities for correct classes.*
  3. *It can handle imbalanced datasets by giving more weight to the minority class during training.*
- *Disadvantages*
  1. *The presence of outliers in the dataset can affect the training process and the final model performance.*
  2. *Lead to overconfidence in the model predictions, where the model assigns high probabilities to incorrect classes as well.*

Hinge Loss:

- The hinge loss is a loss function used for training classifiers, most notably the SVM and other algorithms that involve linear classifiers.
- The hinge loss penalizes the model for making incorrect predictions by the amount of the margin between the predicted score and the true label.
- The hinge loss function is convex, which means it has a unique minimum and can be efficiently optimized using gradient-based methods.



Loss= max(0, 1 - y*f(x))

- *Advantages*
  1. *Hinge loss is less sensitive to outliers than other loss functions*
  2. *Computational efficiency*
  3. *Hinge loss can handle large datasets well, because it only relies on a subset of the data points (i.e., the support vectors) during training.*
- *Disadvantages*
  1. *Not be suitable for non-linear models such as neural networks.*

2. *Has a hyperparameter called the regularization parameter that controls the trade-off between maximizing the margin and minimizing the classification error.*
3. *Hinge loss may not work well with imbalanced datasets*

KL Divergence loss:

- The Kullback-Leibler Divergence score quantifies how much one probability distribution differs from another probability distribution.
- KL divergence is a powerful loss function that can be used to train generative models to produce high-quality samples.
- The KL divergence between two distributions Q and P is often stated using the following notation: KL(P || Q) Where the "||" operator indicates "*divergence*" or Ps divergence from Q.

$$KL(P||Q) = \sum p_i(x) log(\frac{p_i(x)}{q_i(x)})$$

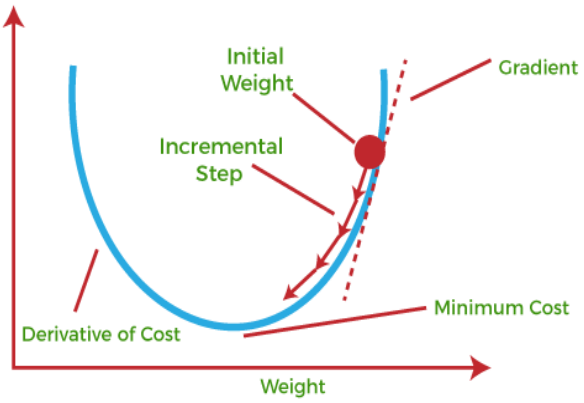
- *Advantages*
  1. Encourages model to capture the entire distribution.
  2. It used to measure the difference between any two probability distributions, not just Gaussian distributions. It can even handle non-gaussian distributions.
- *Disadvantages*
  1. KL divergence is sensitive to outliers in the data, which can lead to the model overfitting to the training data
  2. It requires computing the logarithm of the probability distributions, which can be computationally expensive for high-dimensional data.

Optimization techniques

- Optimization in deep learning refers to the process of adjusting the parameters of a neural network to minimize a specific objective function or loss function.
- The objective function is a measure of how well the neural network is performing on a specific task, such as image classification or natural language processing.
- The goal of optimization is to find the values of the parameters that minimize the objective function, which leads to better performance of the neural network on the task.
- There are several optimization techniques used in deep learning, some of which are as follows:
  - Gradient Descent
  - AdaGrad
  - RMSProp
  - Adam
  - Adadelta

Gradient Descent:

- It is an optimization algorithm used to minimize the loss function of a neural network during training.
- Gradient Descent is a first-order optimization algorithm. It involves taking steps in the opposite direction of the gradient in order to find the global minimum (or local minimum in non-convex functions) of the objective function. The image below provides a great illustration of how Gradient Descent takes steps towards the global minimum of a convex function.
- Gradient Descent starts with a point (w0, w1) in the weight space, and then moves to the neighbouring point that is downhill. This is repeated until we converge on minimum possible loss.
- α is called the learning rate. It can be a fixed constant or it can decay over time as the learning process proceeds.



$w \leftarrow$ *any point in the parameter space*

loop until convergence do

for each w_i *in* w *do*

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(w)$$

- The loss function of univariate regression is a quadratic function. Therefore, the partial derivative are a linear function.
- Consider a simple training example (x, y) and find out the slopes.

$$\frac{\partial}{\partial w_i} Loss(w) = \frac{\partial}{\partial w_i} (y - \hat{y})^2$$
$$= 2\,(y - \hat{y}) \frac{\partial}{\partial w_i} (y - \hat{y})$$
$$= 2\,(y - \hat{y}) \frac{\partial}{\partial w_i} \left(y - (w_0 + w_1 x)\right)$$

- Applying this to both w_0 and w_1, we get

$$\frac{\partial}{\partial w_0} Loss(w) = -2\,(y - \hat{y})$$
$$\frac{\partial}{\partial w_1} Loss(w) = -2\,(y - \hat{y})\,x$$

- When you use these values in the weight updation function, they will become

$$w_0 \leftarrow w_0 + \alpha\,(y - \hat{y})$$
$$w_1 \leftarrow w_1 + \alpha\,(y - \hat{y})\,x$$

- If $\hat{y} > y$,
  - Reduce w_0 a bit
  - Reduce w_1 if x is positive input or increase w_1 if x is a negative input.
- The above weight updation equations are for one training example. For N training examples, we want to minimize the sum of the individual losses for each example.
- The derivative of sum is the sum of the derivatives.

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - \hat{y}_j)$$

$$w_1 \leftarrow w_1 + \alpha \sum_j (y_j - \hat{y}_j)\,x_j$$

- The learning rate is a critical hyperparameter in gradient descent. If the learning rate is too small, the network will converge very slowly, while if it is too large, the network may overshoot the minimum of the loss function and fail to converge. Therefore, finding an appropriate learning rate is an important step in optimizing neural network training.
- **Variants of Gradient Descent**

  1) Batch Gradient Descent:

  The gradient of the loss function is computed with respect to the weights for the entire training dataset, and the weights are updated after each iteration. This provides a more accurate estimate of the gradient, but it can be computationally expensive for large datasets.

  2) Stochastic Gradient Descent (SGD):

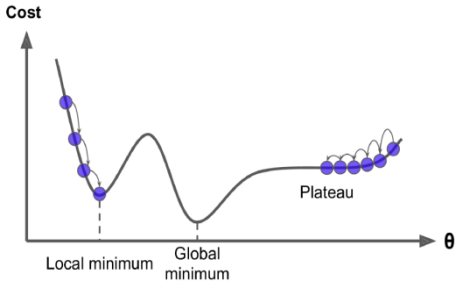
  The gradient of the loss function is computed with respect to a single training example, and the weights are updated after each example. SGD has a lower computational cost per iteration compared to batch gradient descent, but it can be less stable and may not converge to the optimal solution.

3) Mini-Batch Gradient Descent

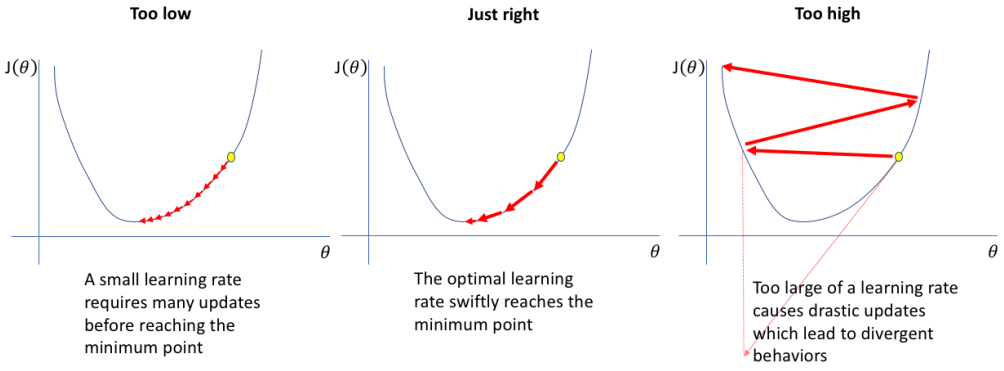
The gradient of the loss function is computed with respect to a small randomly selected subset of the training examples (called a mini-batch), and the weights are updated after each mini-batch. Mini-batch gradient descent provides a balance between the stability of batch gradient descent and the computational efficiency of SGD.

PARAMETERS	BATCH GD ALGORITHM	MINI BATCH ALGORITHM	STOCHASTIC GD ALGORITHM
ACCURACY	HIGH	MODERATE	LOW
TIME CONSUMING	MORE	MODERATE	LESS

- **Challenges** with Gradient Descent are
  - **Local minima:** The loss function of a neural network can have multiple local minima, which can lead to the optimization getting stuck in a suboptimal solution. This can be a problem if the global minimum of the loss function is significantly better than the local minima.
  - **Plateaus and vanishing gradients:** In deep neural networks with many layers, the gradients can become very small (vanishing gradients) or very large (exploding gradients) as they propagate through the layers. This can make the optimization process very slow or even stall altogether, especially in the presence of plateaus in the loss function.



  - **Overfitting:** Gradient descent can overfit to the training data, leading to poor generalization performance on new, unseen data. This can occur if the network is too complex or if the training data is insufficient or noisy.
  - **Hyperparameter tuning**: Gradient descent has several hyperparameters, including the learning rate, the batch size, and the momentum, that must be tuned to achieve optimal performance. Finding the optimal set of hyperparameters can be time-consuming and requires careful experimentation.



  - **Computation time:** Gradient descent requires the computation of the gradient of the loss function with respect to the model parameters, which can be computationally expensive for large datasets and complex models. This can make training a neural network very time-consuming, especially if multiple epochs are required.

AdaGrad:

- Adaptive Gradient Algorithm (Adagrad) is an algorithm for gradient-based optimization.
- The learning rate is adapted component-wise to the parameters by incorporating knowledge of past observations.
- It uses different learning rate for each and every neuron in the layer, throughout the neural network.
- Equation for Adagrad:

$$w_t = w_{t-1} - \eta'_t . \frac{\partial L}{\partial w_{t-1}}$$

Where ,

$$\eta'_t = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_i}\right)^2$ -- Adaptive Moment Estimation

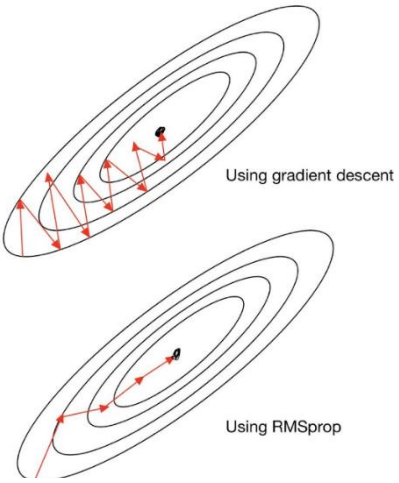
ϵ – Positive constant

- It starts with big learning rate, slowly the learning will reduce for iterations and then finally it converges.
- The idea behind Adagrad is normalize the gradient for each parameter by dividing them by a running sum of squared gradient.
- This normalization helps to reduce the learning rate for frequently occurring parameters and increase the learning rate for infrequently occurring parameter.
- The Adagrad algorithm can be summarized in the following steps:
    - Initialize the sum of squared gradients to zero for each parameter.
    - For each iteration:
        - Compute the gradients for the current mini-batch of training examples.
        - Update the sum of squared gradients for each parameter by adding the squared gradients of the current mini-batch.
        - Compute the learning rate for each parameter by dividing the initial learning rate by the square root of the sum of squared gradients for that parameter.
        - Update each parameter by subtracting the product of the learning rate and the gradient for that parameter.

- Adagrad is particularly effective for sparse data, where many of the gradients are zero or very small.

RMSprop:

- **Root Mean Squared Propagation**, or **RMSProp**, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter.
- It is a variant of the stochastic gradient descent (SGD) algorithm that aims to accelerate the convergence of the training process.
- The key idea behind RMSprop is to adjust the learning rate for each parameter based on the magnitude of the average of recent gradients for that parameter.
- Specifically, RMSprop uses a moving average of the squared gradients to scale the learning rate. This helps to prevent the learning rate from being too large or too small, and also helps to avoid oscillations in the update process.



RMSProp

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Backpropagation: Refer in slides

Note:

- The bias error is an error from wrong assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs. This is called **underfitting.**
- The variance is an error from sensitivity to small fluctuations in the training set. High variance may result in modeling the random noise in the training data. This is called **overfitting.**

Training Deep Models: Training deep neural networks involves optimizing the parameters of the network so that it can learn to make accurate predictions or classifications on a given dataset. The following steps are typically involved in training deep neural networks:

- **Data Preparation:** The first step is to prepare the data for training. This includes splitting the data into training, validation, and test sets, normalizing the data, and encoding categorical features.
- **Model Selection:** Next, the type of neural network architecture to use must be selected. This could be a feedforward network, a convolutional network, a recurrent network, or some combination of these.
- **Initialization:** The initial values of the model's parameters need to be set. This is usually done randomly or by using pre-trained weights.
- **Forward Propagation:** The input data is fed forward through the network, and the output is computed.
- **Loss Calculation:** The difference between the predicted output and the actual output is computed, and a loss function is used to measure how well the network is doing.
- **Backward Propagation:** The gradients of the loss function with respect to each of the parameters in the network are computed.
- **Optimization:** The gradients are used to update the network's parameters, usually using an optimization algorithm such as stochastic gradient descent.
- **Validation:** The network's performance is evaluated on the validation set, and the process is repeated until the performance on the validation set stops improving.
- **Testing:** Finally, the performance of the network is evaluated on the test set to see how well it generalizes to new data.

The training process can be time-consuming, especially for large datasets and complex networks, and often requires significant computational resources. However, with careful tuning of hyperparameters and the use of techniques such as early stopping and regularization, it is possible to train deep neural networks that achieve state-of-the-art performance on a wide range of tasks.

Regularization:

- Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well, and thus prevent the overfitting.
- ***Regularization as "any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."***
- Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias.
- Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty Ω(θ) to the objective function J. We denote the regularized objective function by

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\, \Omega(\theta)$$

  Where α belongs [0, ∞) is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J. Setting α to 0 results in no regularization. Larger values of α correspond to more regularization
- Types of regularization techniques
  - L1 regularization
  - L2 regularization
  - Dropout regularization
  - Data Augmentation
  - Early stopping

L1 regularization

- Also known as L1 norm or Lasso combats overfitting by shrinking the parameters towards 0. This makes some features obsolete.
- The q value is 1 in L1 regularization
- we simply use another regularization term **Ω**. This term is the sum of the absolute values of the weight parameters in a weight matrix:

$$\Omega(W) = ||W||_1 = \sum_i \sum_j |w_{ij}|$$

- we multiply the regularization term by alpha and add the entire thing to the loss function.

$$\hat{\mathcal{L}}(W) = \alpha||W||_1 + \mathcal{L}(W)$$

- The derivative of the new loss function leads to the following expression, which the sum of the gradient of the old loss function and sign of a weight value times alpha.

$$\nabla_W \hat{\mathcal{L}}(W) = \alpha sign(W) + \nabla_W \mathcal{L}(W)$$

L2 Regularization:

- L2 regularization, or the L2 norm, or Ridge combats overfitting by forcing weights to be small, but not making them exactly zero.
- Also commonly known as weight decay.
- When L2 regularization is used, the less significant features also have some influence over the final prediction.

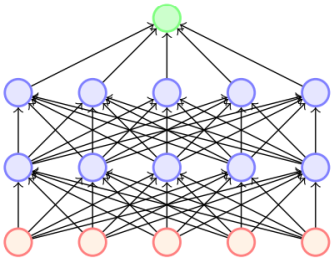
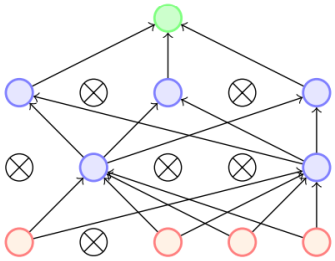
$$\Omega(W) = ||W||_2^2 = \sum_i \sum_j w_{ij}^2$$

- The regularization term Ω is defined as the Euclidean Norm (or L2 norm) of the weight matrices, which is the sum over all squared weight values of a weight matrix.
- The regularization term is weighted by the scalar alpha divided by two and added to the regular loss function that is chosen for the current task.

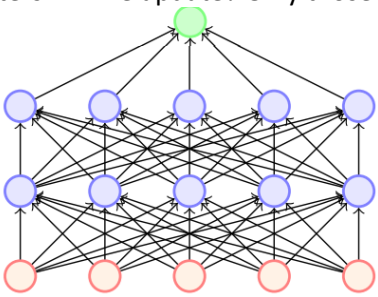
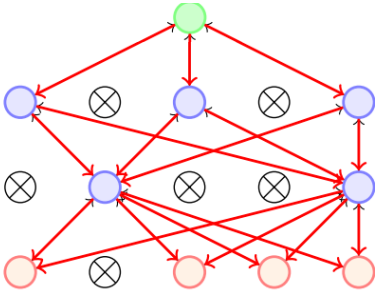
$$\hat{\mathcal{L}}(W) = \frac{\alpha}{2}||W||_2^2 + \mathcal{L}(W) = \frac{\alpha}{2}\sum_i \sum_j w_{ij}^2 + \mathcal{L}(W)$$

Dropout:

- Dropout is a technique which addresses overfitting issue.
- Effectively it allows training several neural networks without any significant computational overhead.
- Also gives an efficient approximate way of combining exponentially many different neural networks.

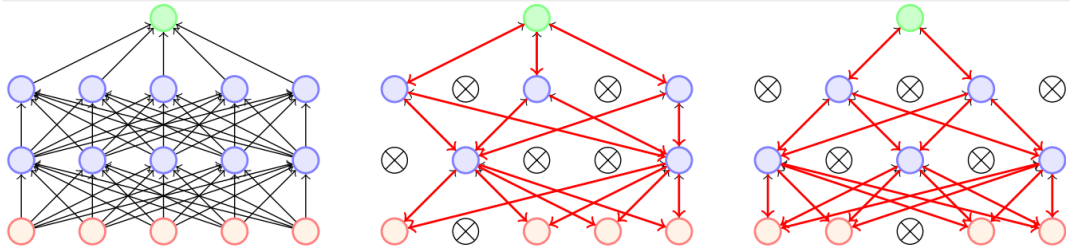



- Dropout refers to dropping out units.
- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network
- Each node is retained with a fixed probability (typically p = 0.5) for hidden nodes and p = 0.8 for visible nodes
- We initialize all the parameters (weights) of the network and start training
- For the first training instance (or mini-batch), we apply dropout resulting in the thinned network
- We compute the loss and backpropagate
- Which parameters will we update? Only those which are active

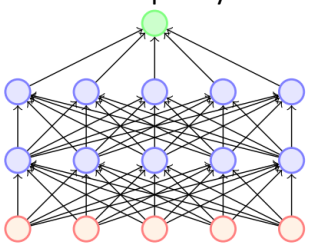



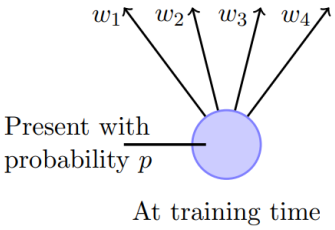
- For the second training instance (or mini-batch), we again apply dropout resulting in a different thinned network
- We again compute the loss and backpropagate to the active weights

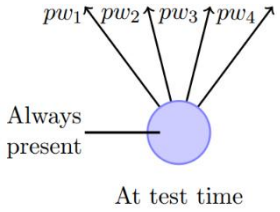
- If the weight was active for both the training instances then it would have received two updates by now
- If the weight was active for only one of the trainings instances, then it would have received only one updates by now
- Each thinned network gets trained rarely (or even never) but the parameter sharing ensures that no model has untrained or poorly trained parameters



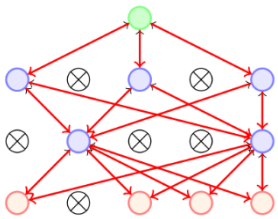
- For the second training instance (or mini-batch), we again apply dropout resulting in a different thinned network
- We again compute the loss and backpropagate to the active weights
- If the weight was active for both the training instances then it would have received two updates by now
- If the weight was active for only one of the training instances then it would have received only one updates by now
- Each thinned network gets trained rarely (or even never) but the parameter sharing ensures that no model has untrained or poorly trained parameters

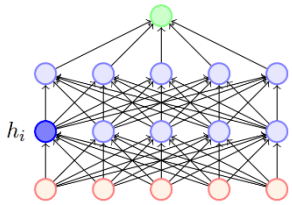






- Impossible to aggregate the outputs of 2^n thinned networks.
- Instead, we use the full Neural Network and scale the output of each node by the fraction of times it was on during training.
- Dropout essentially applies a masking noise to the hidden units
- Prevents hidden units from co-adapting
- Essentially a hidden unit cannot rely too much on other units as they may get dropped out any time
- Each hidden unit must learn to be more robust to these random dropouts.





Here is an example of how dropout helps in ensuring redundancy and robustness. Suppose h_i learns to detect a face by firing on detecting a nose. Dropping h_i then corresponds to erasing the information that a nose exists. The model should then learn another h_i which redundantly encodes the presence of a nose Or the model should learn to detect the face using other features

Early Stopping:

- Early stopping is a popular technique used in deep learning to prevent overfitting of a model by monitoring its performance during training and stopping it when it starts to overfit the training data.
- The basic idea behind early stopping is to find the point during training where the model has learned the general pattern in the data but has not yet overfit the noise of the training set. This point is typically determined by looking for the point where the performance on the validation set starts to plateau or decrease.
- In practice, early stopping is implemented by training the model for a fixed number of epochs or until the validation performance reaches a certain threshold.
- If the validation performance does not improve after a certain

number of epochs, training is stopped and the model with the best validation performance is selected as the final model.

- **Early stopping – Algorithm**
  - Divide the dataset into three parts: training set, validation set, and test set. The test set is used to evaluate the final performance of the model.

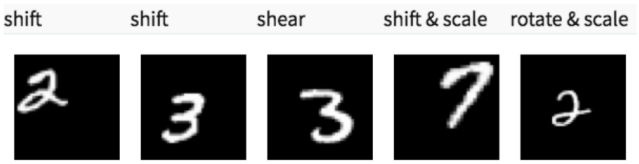
  - Initialize the model parameters, choose a loss function, and select an optimizer.
  - Train the model on the training set for a fixed number of epochs, evaluating the performance on the validation set at regular intervals.
  - If the validation performance does not improve for a certain number of consecutive epochs, stop the training and select the model with the best validation performance.
  - Evaluate the selected model on the test set to obtain an estimate of its performance on unseen data.

```
from keras.callbacks import EarlyStopping
EarlyStopping(monitor='val_err', patience=5)
```

    - Here, **monitor** denotes the quantity that needs to be monitored and '**val_err**' denotes the validation error.
    - **Patience** denotes the number of epochs with no further improvement after which the training will be stopped.
    - For better understanding, let's take a look at the above image again. After the dotted line, each epoch will result in a higher value of validation error. Therefore, 5 epochs after the dotted line (since our patience is equal to 5), our model will stop because no further improvement is seen.

Augmentation:

- Data augmentation can be used as a regularization technique in deep learning to prevent overfitting and improve the generalization performance of the model.
- Data augmentation works as a regularization technique by introducing small variations in the training data, which can help the model to generalize better to new data.
- The simplest way to reduce overfitting is to increase the size of the training data.
- In machine learning, we were not able to increase the size of training data as the labeled data was too costly.
- By augmenting the data, the model is exposed to a wider variety of examples, which helps it to learn more robust features and patterns that are useful for classification or prediction tasks.




```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

- Common examples of data augmentation include flipping images horizontally or vertically, randomly cropping images, adding noise or distortion, and changing the brightness or contrast.
- These modifications create variations of the original data, which can help prevent the model from memorizing specific examples.
- It can be used in combination with other regularization techniques, such as weight decay and dropout, to further improve the model's performance and prevent overfitting.