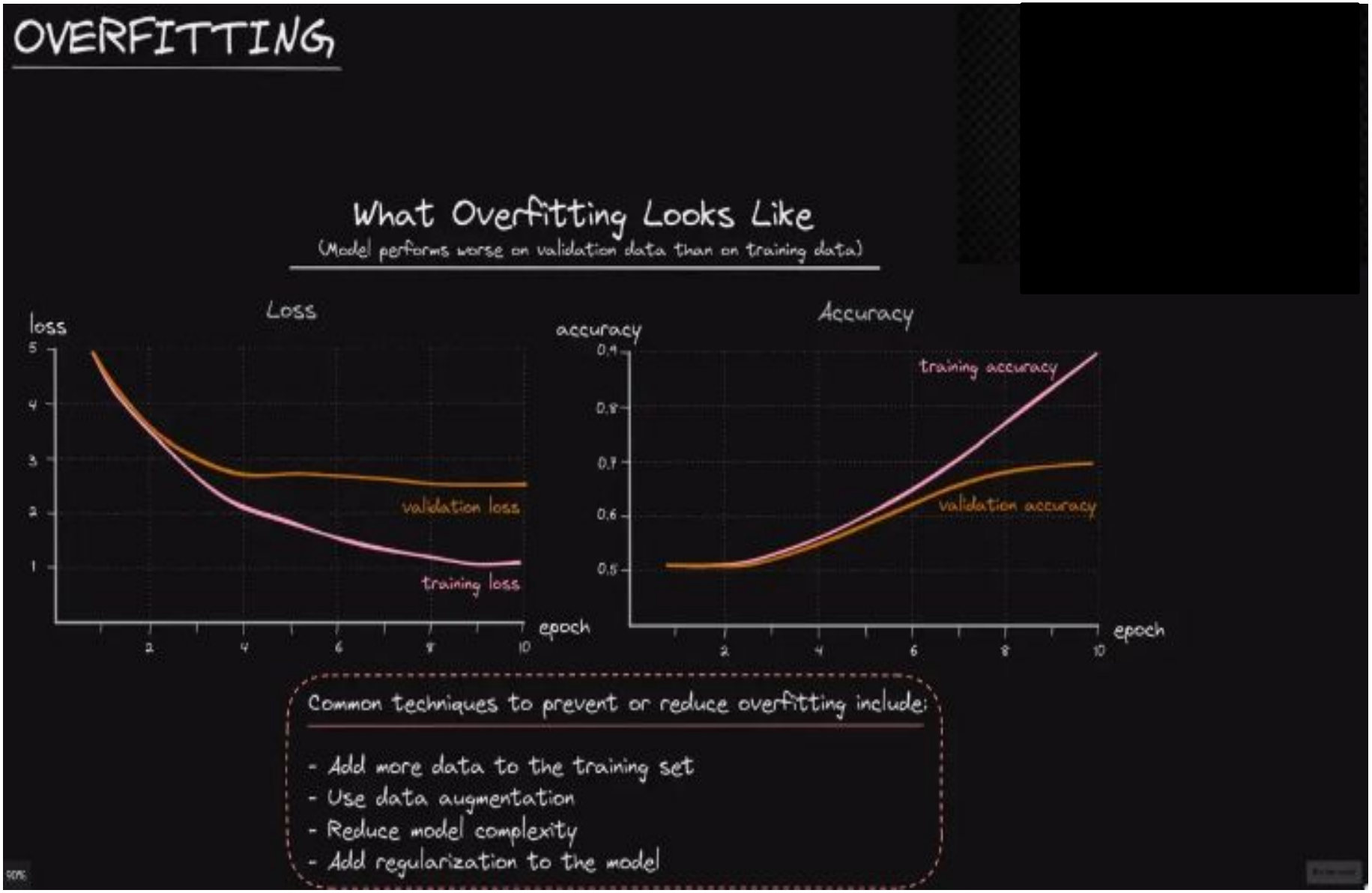
ANN AND DEEP LEARNING (CS636)

BY: Nidhi S. Periwal,
Teaching Assistant,
COED, SVNIT, Surat

OVERFITTING



Regularization

- When learning model, emulates the training data closely and performs very well w.r.t. to training data but fails to perform w.r.t. test data, the model is said to be over-fitted model.
- Weights in NN are crucial parameter which lead to the performance of model on test data.
- Hence, if the weights are rightly assigned, the network will generalized well and give right prediction of test data.
- **However, in case of over-fitting, the weight values are assigned in such a way that it gives accurate predictions for training data but for test data predictions are grossly inaccurate.**
- **One potential symptom of such an over-fitted network is when certain values have excessively large values.**
- **A small change in those inputs may lead to a large change in the value of the output, which is not a good example of generalization model.**

Regularization

- **It is used to prevent the high value of weights.**
- **The learning algorithm should try to keep the weights small, which can be achieved by regularization.**
- **Regularization attempts to penalize large weights by increasing loss values.**
- The optimization algorithm **try to minimize the loss – they try to decrease the weights of the network.**
- Since, **the intent is to penalize higher weights & make them smaller & more regular weights, the process is known as regularization.**

Regularization

Penalty-based regularization

- Penalize the *parameters* of the neural network.
- Penalty-based regularization is the most common approach for reducing overfitting.

$$\hat{y} = \sum_{i=0}^{d} w_i x^i$$

- Eqn: Single-layer network with *d* inputs and a single bias neuron with weight w_0 in order to model this prediction.
- The i^{th} input is x_i. This neural network uses linear activations, and the squared loss function for a set of training instances (*x, y*) from data set *D* can be defined as follows:

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2$$

- **A large value of *d* tends to increase overfitting.**
- **One possible solution to this problem is to reduce the value of *d*.** In other words, using a model with *economy in parameters* leads to a simpler model.
- Reducing ***d* to 1 creates a linear model** that has fewer degrees of freedom and tends to fit the data in a similar way over different training samples. **However, doing so does lose some expressivity when the data patterns are indeed complex**.
- In other words**, oversimplification reduces the expressive power of a neural network, so that it is unable to adjust sufficiently to the needs of different types of data sets.**

- Instead of reducing the number of parameters in a hard way, one can use a ***soft* penalty on the use of parameters**
- **Large (absolute) values of the parameters are penalized more than small values, because small values do not affect the prediction significantly.**
- The most common choice is *L*2-regularization, which is also referred to as ***Tikhonov*** *regularization*.
- It uses squared norm penalty, is the most common approach for regularization.

L_2-Regularization

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2 + \lambda \cdot \sum_{i=0}^{d} w_i^2$$

- Increasing or decreasing the value of $\boldsymbol{\lambda}$ **reduces the softness of the penalty.**
- One advantage of this type of parameterized penalty is that **one can tune this parameter for optimum performance on a portion of the training data set that is not used for learning the parameters.** This type of approach is referred to as *model validation*.

- For any given weight w_i in the neural network, the updates are defined by gradient descent.

$$w_i \Leftarrow w_i - \alpha \frac{\partial L}{\partial w_i}$$

- Here, α is the learning rate. The use of *L2*-regularization is roughly equivalent to the use of decay imposition after each parameter update

$$w_i \Leftarrow w_i(1 - \alpha\lambda) - \alpha \frac{\partial L}{\partial w_i}$$

- The update above first multiplies **the weight with the decay factor $(1 - \alpha\lambda)$,** and then uses the gradient-based update.
- If we assume that the initial values of the weights are close to 0. One can view weight decay as a kind of **forgetting mechanism**, which brings the weights closer to their initial values. This ensures that only the repeated updates have a significant effect on the absolute magnitude of the weights.
- **A forgetting mechanism prevents a model from *memorizing* the training data,** because only significant and repeated updates will be reflected in the weights.

L_1-Regularization

- L_1-regularization uses a penalty on the sum of the absolute magnitudes of the coefficients. The new objective function is as follows:

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2 + \lambda \cdot \sum_{i=0}^{d} |w_i|_1$$

- Weight update equation:

$$w_i \Leftarrow w_i - \alpha\lambda s_i - \alpha\frac{\partial L}{\partial w_i}$$

The value of s_i, which is the partial derivative of $|w_i|$ (with respect to w_i), is as follows:

$$s_i = \begin{cases} -1 & w_i < 0 \\ +1 & w_i > 0 \end{cases}$$

- For the rare cases in which the value w_i is exactly 0, one can omit the regularization and simply set s_i to 0

L2 vs L1

- *L*2-regularization uses multiplicative decay as a forgetting mechanism, whereas *L*1-regularization uses additive updates as a forgetting mechanism.
- In both cases, the regularization portions of the updates tend to move the coefficients closer to 0.
- From an accuracy point of view, *L*2-regularization usually outperforms *L*1-regularization

- An interesting property of *L*1-regularization is that it creates *sparse* solutions in which the vast majority of the values of w_i are 0s (after ignoring computational errors).
- If the value **of w_i is zero for a connection incident on the input layer, then that particular input has no effect on the final prediction. In other words, such an input can be *dropped* and the *L*1-regularizer acts as a feature selector.**
- Therefore, one can use *L*1-regularization to estimate which features are predictive to the application at hand.
- These connections can be dropped, which results in a sparse neural network. Such sparse neural networks can be useful in cases where one repeatedly performs training on the same type of data set, but the nature and broader characteristics of the data set do not change significantly with time.
- Since **the sparse neural network will contain only a small fraction of the connections in the original neural network, it can be retrained much more efficiently whenever more training data is received**.

Regularization

- **L_1/L_2 Regularization:**
  - **If loss function is assumed to be L(w,b), the objective of any optimization algorithm is to minimize the value of loss function.**
  - For **L_1 /LASSO/ Linear or logistics regression , the sum of absolute values of the weights is added to the loss function.**
  - For L_2/ Ridge regression for linear or logistic regression/, the sum of **squared** values of weights is added to the loss function.

L_1 regularization (or L_1 norm): $L'(w,b) = L(w,b) + \lambda w_1$

L_2 regularization (or L_2 /Euclidean norm): $L'(w,b) = L(w,b) + \lambda w_2^2$

  - Lambda is regularization parameter, which is hyper-parameter.

Regularization

- L_1/L_2 Regularization:
  - As the optimization algorithm will try to reduce the value L'(w,b), it will also try to bring down the value of penalty term.
  - Hence, setting a larger value for regularization parameter will push down the value of weight parameter w.
  - It is also possible to combine both L_1 and L_2 regularization.

$$L'(w,b) = L(w,b) + \lambda_1 w_1 + \lambda_2 w_2^2$$

- Extreme forms of overfitting are referred to as *memorization.*
- The ability of a learner to provide useful predictions for instances it has not seen before is referred to as *generalization.*

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_i = w_i - \eta \frac{d}{dw_i}(loss(w) + regularizer(w, b))$$

$$w_j = w_j + \eta \sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta \lambda w_j$$

The update

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta \lambda w_j$$

learning rate

direction to update

constant: how far from wrong

regularization

What effect does the regularizer have?

The update

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta \lambda w_j$$

learning rate

direction to update

constant: how far from wrong

regularization

If w_j is positive, reduces w_j
If w_j is negative, increases w_j

} moves w_j towards 0

L1 regularization

$$\text{argmin}_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) + \|w\|$$

$$\frac{d}{dw_j} objective = \frac{d}{dw_j} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) + \lambda \|w\|$$

$$= -\sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) + \lambda sign(w_j)$$

L1 regularization

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta\lambda sign(w_j)$$

learning rate

direction to update

constant: how far from wrong

regularization

What effect does the regularizer have?

L1 regularization

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta\lambda sign(w_j)$$

learning rate

direction to update

constant: how far from wrong

regularization

If w_j is positive, reduces by a constant
If w_j is negative, increases by a constant

moves w_j towards 0 ***regardless of magnitude***

Regularization with p-norms

L1:

$$w_j = w_j + \eta(loss\_correction - \lambda sign(w_j))$$

L2:

$$w_j = w_j + \eta(loss\_correction - \lambda w_j)$$

Lp:

$$w_j = w_j + \eta(loss\_correction - \lambda c w_j^{p-1})$$

How do higher order norms affect the weights?

Regularizers summarized

L1 is popular because it tends to result in sparse solutions (i.e. lots of zero weights)

- However, it is not differentiable, so it only works for gradient descent solvers

L2 is also popular because for some loss functions, it can be solved directly (no gradient descent required, though often iterative solvers still)

Lp is less popular since they don't tend to shrink the weights enough

The other loss functions

Without regularization, the generic update is:

$$w_j = w_j + \eta y_i x_{ij} c$$

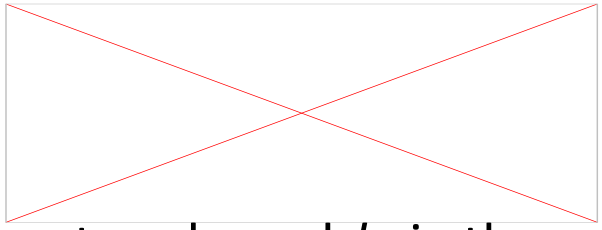
where

$c = \exp(-y_i(w \cdot x_i + b))$	exponential
$c = 1[yy' < 1]$	hinge loss

$$w_j = w_j + \eta(y_i - (w \cdot x_i + b)x_{ij})$$ squared error

Penalizing Hidden Units

- An approach is to **penalize the *activations* of the neural network,** so that only a small subset of the neurons are activated for any given data instance.
- **Even though the neural network might be large and complex only a small part of it is used for predicting any given data instance.**
- The simplest way to achieve sparsity is to impose an $L1$-penalty on the hidden units. The original loss function L is modified to the regularized loss function L as follows:



- M is the total number of units in the network, and h_i is the value of the i^{th} hidden unit. The regularization parameter is denoted by λ.
- In many cases, a single *layer* of the network is regularized, so that a sparse feature representation can be extracted from the activations of that particular layer

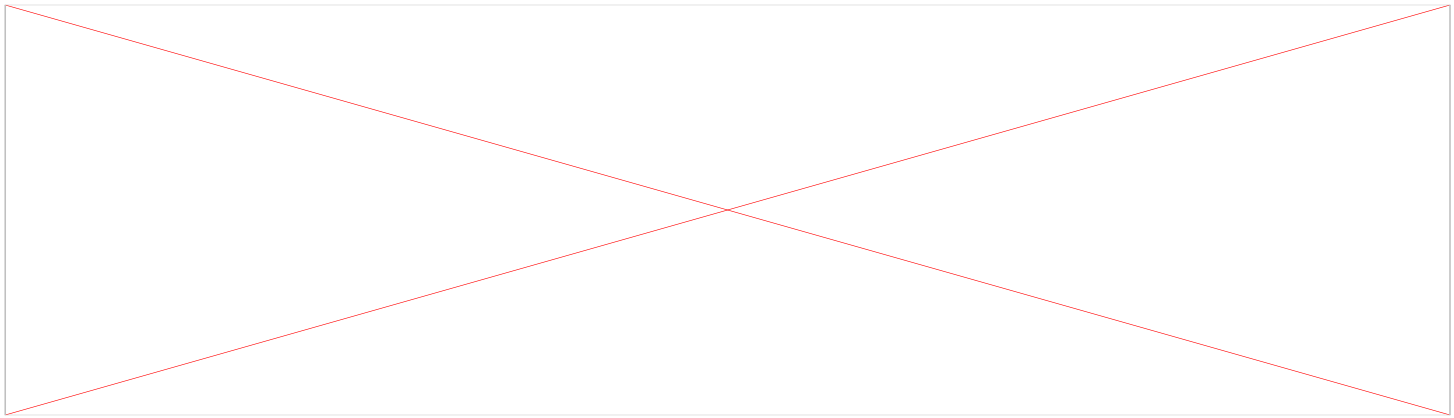
Randomized Connection Dropping

- The random dropping of connections between different layers in a multilayer neural network often leads to diverse models in which different combinations of features are used to construct the hidden variables.
- **Edge Sampling-** The dropping of connections between layers does tend to create less powerful models because of the addition of constraints to the model-building process.
- However, since different random connections are dropped from different models, the predictions from different models are very diverse.
- The averaged prediction from these different models is often highly accurate.
- It does not share any weights between ensemble components.

- Randomized connection dropping can be used for any type of predictive problem and not just classification. For example, the approach has been used for outlier detection with autoencoder ensembles.
- Autoencoders can be used for outlier detection by estimating the reconstruction error of each data point.
- Multiple autoencoders with randomized connections, and then aggregates the outlier scores from these different components in order to create the score of a single data point.
- This approach

Dropout

- The strategy applied for dropout is simple.
- Dropout means that during training with some probability **P** a neuron of the neural network gets turned off during training.
- Assume on the left side we have a feedforward neural network with no dropout.
- Using dropout with let's say a probability of **P=0.5** that a random neuron gets turned off during training would result in a neural network on the right side.



Dropout

- *Dropout* is a method that **uses node sampling** instead of edge sampling in order to create a neural network ensemble
- **If a node is dropped, then all incoming and outgoing connections from that node need to be dropped as well. The nodes are sampled only from the input and hidden layers of the network.**
- The nodes are sampled only from the input and hidden layers of the network.
- Note that, sampling the output node(s) would make it impossible to provide a prediction and compute the loss function.
- In some cases, the input nodes are sampled with a different probability than the hidden nodes.
- Therefore, if the full neural network contains M nodes, then the total number of possible sampled networks is 2^M.
- *Dropout* combines node sampling with weight sharing.

Dropout

- The training process proceeds using the following steps, which are repeated again and again in order to cycle through all of the training points in the network:
  - Sample a neural network from the base network. The input nodes are each sampled with probability p_i, and the hidden nodes are each sampled with probability p_h. Furthermore, all samples are independent of one another. When a node is removed from the network, all its incident edges are removed as well.
  - Sample a single training instance or a mini-batch of training instances.
  - Update the weights of the retained edges in the network using backpropagation on the sampled training instance or the mini-batch of training instances
- It is common to exclude nodes with probability between 20% and 50%.

Dropout

- In the *Dropout* method, thousands of neural networks are sampled with shared weights, and a tiny training data set is used to update the weights in each case.
- ***Weight scaling inference rule:*** Forward propagation can be performed on only the base network (with no dropping) after re-scaling the weights.
- **The basic idea is to multiply the weights going out of each unit with the probability of sampling that unit.**
- By using this approach, the expected output of that unit from a sampled network is captured. This rule is referred to as the *weight scaling inference rule*.

Dropout

- The weight scaling inference rule is exact for many types of networks with linear activations, although the rule is not exactly true for networks with nonlinearities.
- In practice, the rule tends to work well across a broad variety of networks. Since most practical neural networks have nonlinear activations, the weight scaling inference rule of *Dropout* should be viewed as a heuristic rather than a theoretically justified result.
- **The main effect of *Dropout* is to incorporate regularization into the learning procedure.**
- By dropping both input units and hidden units, ***Dropout* effectively incorporates noise into both the input data and the hidden representations.**
- The nature of this noise can be viewed as a kind of masking noise in which some inputs and hidden units are set to 0. Noise addition is a form of regularization.

Dropout

- *Dropout* **prevents a phenomenon referred to as *feature co-adaptation* from occurring between hidden units.**
- Since the effect of *Dropout* is a masking noise that removes some of the hidden units, **this approach forces a certain level of redundancy** between the features learned at the different hidden units. This type of redundancy leads to increased robustness.
- *Dropout* is a regularization method, it reduces the expressive power of the network. Therefore, one needs to use larger models and more units in order to gain the full advantages of *Dropout*.

Regularization

Dropout

- In this case, you can observe that approximately half of the neurons are not active and are not considered as a part of the neural network. And as you can observe the neural network becomes simpler.
- **A simpler version of the neural network results in less complexity that can reduce overfitting. The deactivation of neurons with a certain probability P is applied at each forward propagation and weight update step.**