Babysitting the Neural Network

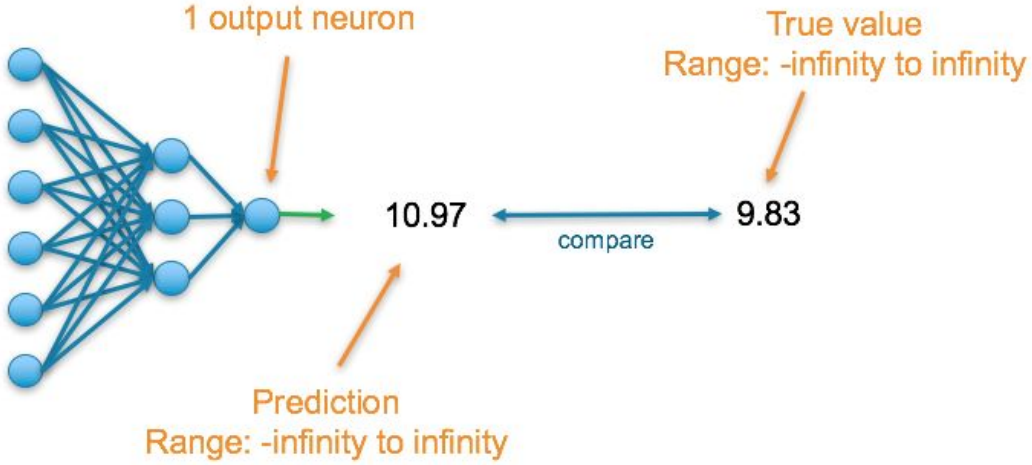
Loss function/Cost function

It is a method of evaluating how well the specific algorithm models data.

It is the difference between the actual value and the predicted value.

As part of the optimization algorithm, the error for the current state of the model must be estimated repeatedly. This requires the choice of an error function, conventionally called a **loss function. There are basically two types of loss functions.**

Regression loss functions.
Classification loss functions.



https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8

Regression Loss function

A Regression problem involves predicting the real-valued quantity.

- **Output Layer Configuration**: One node with a linear activation unit.
- **Loss Function**: Regression loss function

Types of Regression-based loss functions:

1) Mean Absolute loss function (MAE)
2) Mean Square loss function (MSE)
3) Root Mean Square loss function (RMSE)
4) Huber loss (HL)

Mean Square Loss function

It is measured as the average of squared difference between predictions and actual observations.

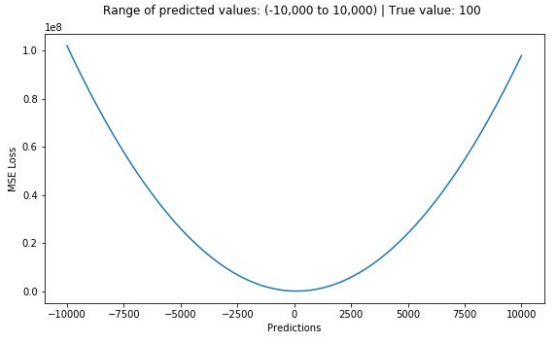
It's only concerned with the average magnitude of error irrespective of their direction

It works well when the target variable is normally distributed

It is highly affected by outliers

We take the square root of MSE (called RMSE) to reduce the effect of higher loss for outliers

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$



https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

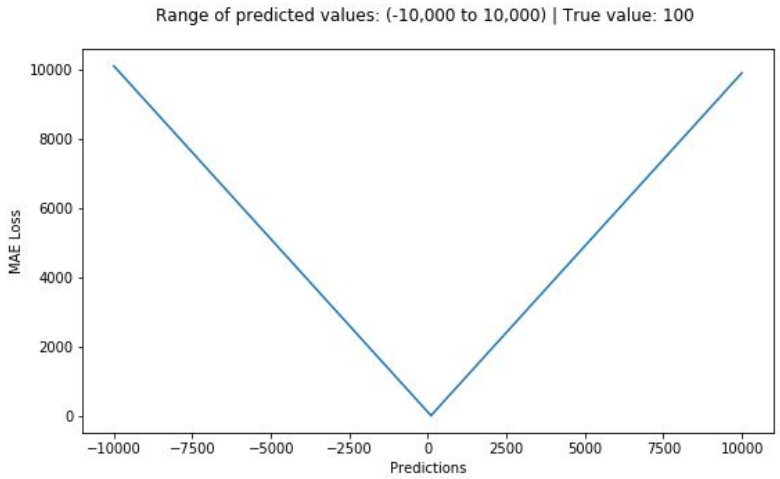
Mean Absolute Error

It is the sum of the absolute differences between our target and predicted variables.

So it measures the average magnitude of errors in a set of predictions, without considering their directions.

It is less affected by outliers than the Mean Square Loss function.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - y_i^p|}{n}$$

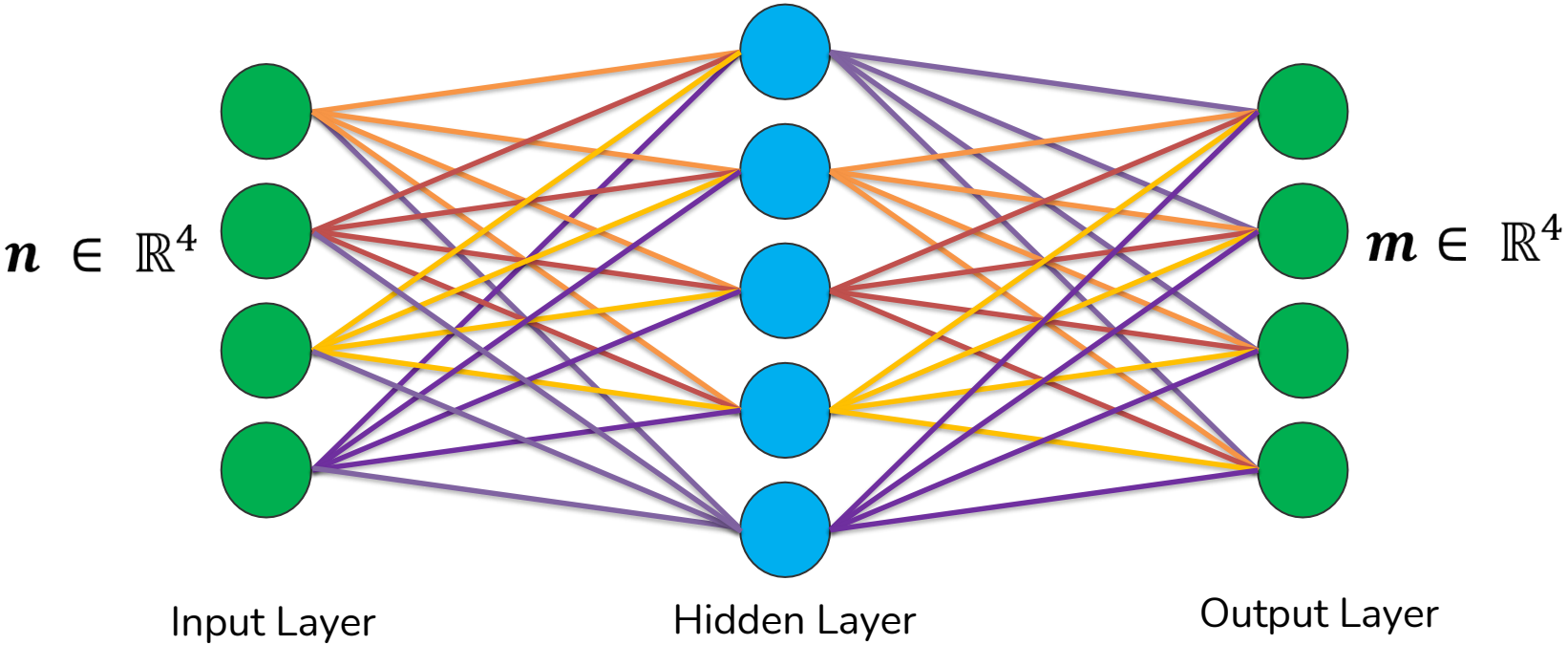


https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

Weight Initialization

What happens when W=0 init is used?

Total Parameters = 29



Different Cases

1. **Initialize all weights with 0** - Your network will not learn as all the weights are same.

2. **Initialize with random numbers** - Works okay for small networks (similar to our two layer MNIST classifier), but it may lead to distributions of the activations that are not homogeneous across the layers of the network.

1. Initializing all weights to 0

- This makes your model equivalent to a linear model.
- When you set all weight to 0, the derivative with respect to loss function is the same for every w in every layer, thus, all the weights have the same values in the subsequent iteration.
- This makes the hidden units symmetric and continues for all the n iterations you run.
- Thus setting weights to zero makes your network no better than a linear model.

Refer -
https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94

Food for thought: What if we were to initialize all weights to a random constant, ex 10?

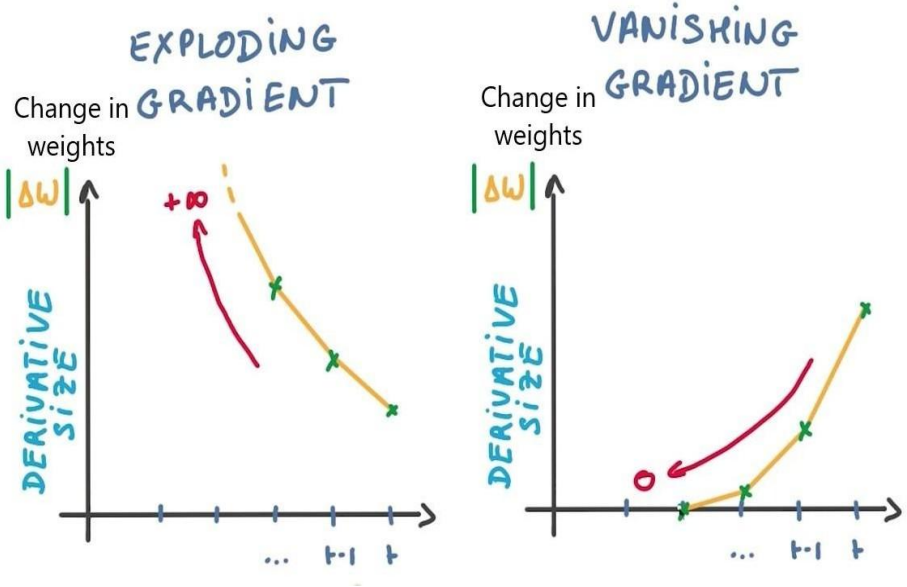
2. Initializing weights randomly

Initializing weights randomly, following standard normal distribution (np.random.randn(n, n-1) in Python) while working with a (deep) neural network can potentially lead to 2 issues — vanishing gradients or exploding gradients.

Vanishing Gradient Descent

Vanishing gradients — In case of deep networks, for any activation function, *abs(dW)* will get smaller and smaller as we go backwards with every layer during back propagation. The earlier layers are the slowest to train in such a case.

The weight update is minor and results in slower convergence. This makes the optimization of the loss function slow. In the worst case, this may completely stop the neural network from training further.

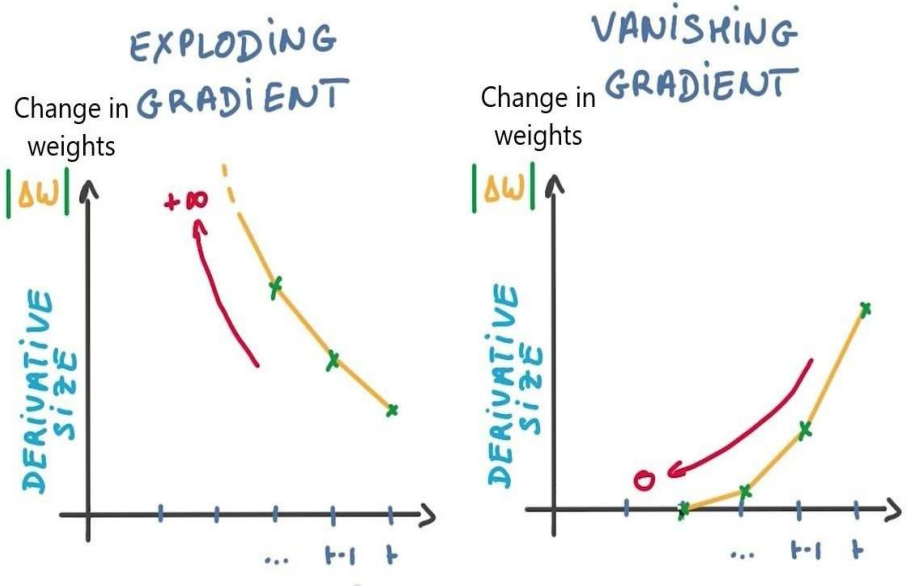


Exploding Gradient Descent

Exploding gradients —This is the exact opposite of vanishing gradients. Consider you have non-negative and large weights and small activations A (as can be the case for sigmoid(z)).

This may result in oscillating around the minima or even overshooting the optimum again and again and the model will never learn!

Another impact of exploding gradients is that huge values of the gradients may cause number overflow resulting in incorrect computations or introductions of NaN's. This might also lead to the loss taking the value NaN.



Best Practices

1. Use ReLU/Leaky ReLU as the activation function, as it is relatively robust to the vanishing/exploding gradient issue (especially for networks that are not too deep).

2. Xavier Initialization - we want to initialize the weights with random values that are not “too small” and not “too large.”

$$\sqrt{\frac{2}{Size\ of\ Previous\ Layer}}$$

3. He Initialization $$\sqrt{\frac{2}{Size\ of\ Previous\ Layer + Size\ of\ Current\ Layer}}$$

Proper Initialization is an active area of research

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks
by Saxe et al, 2013

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification
by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks
by Krähenbühl et al., 2015

All you need is a good init
by Mishkin and Matas, 2015

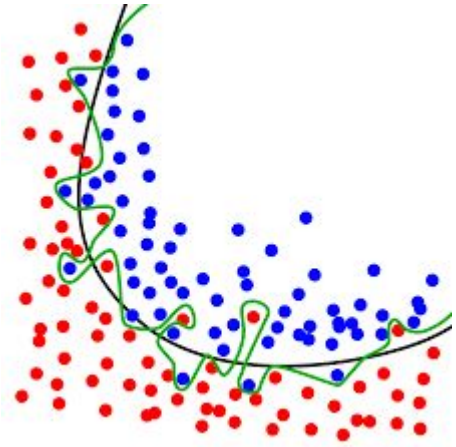
Centered Weight Normalization in Accelerating Training of Deep Neural Networks
by Huang et al., 2017

Adjusting for Dropout Variance in Batch Normalization and Weight Initialization:
by Hendrycks et al., 2017
...

Overfitting in ANN

Neural networks are prone to overfitting because of the larger number of parameters.

ANN is able to model higher order and complex functions which makes it more prone to overfitting .

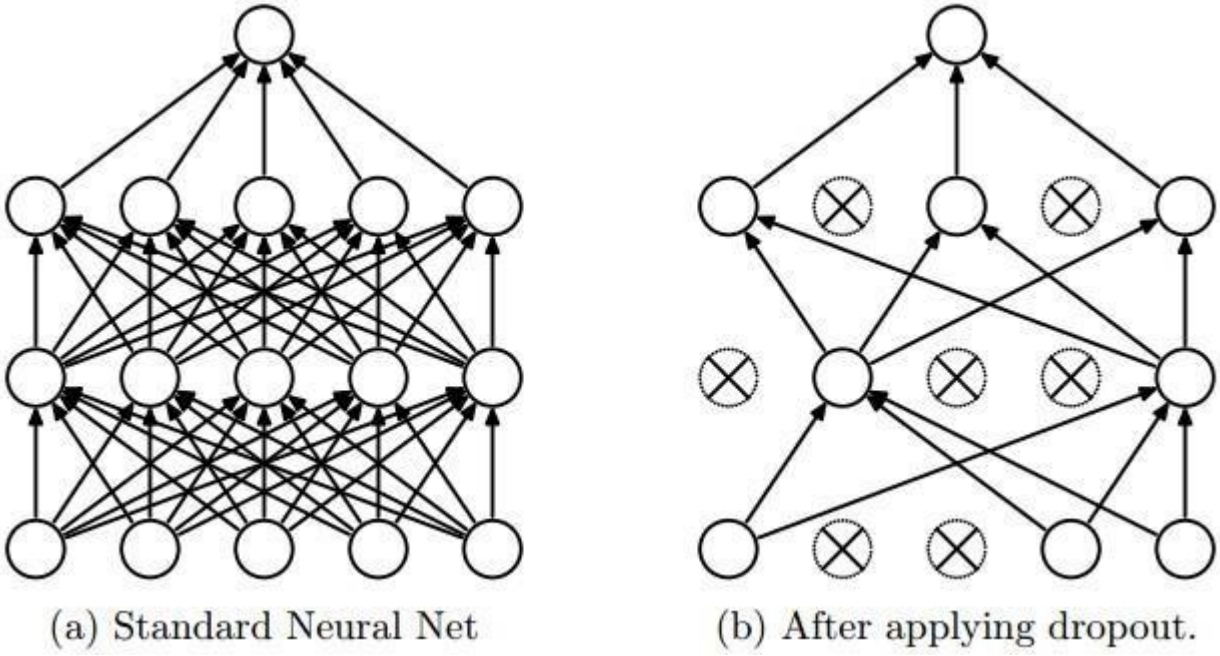


https://en.wikipedia.org/wiki/Overfitting

Solution to overfitting: Regularization

Regularization: Dropout

"During training, randomly set some neurons to zero in the forward pass"



(a) Standard Neural Net

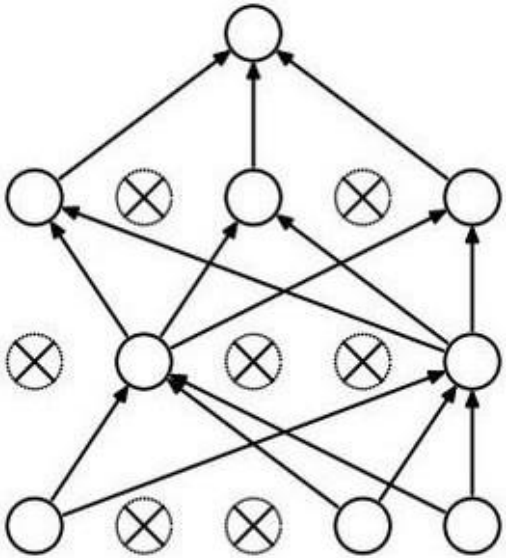
(b) After applying dropout.

Why do we need Dropout?

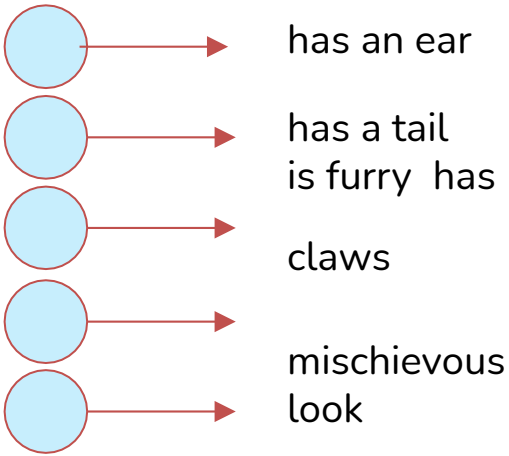
The answer to this question is "to prevent over-fitting".

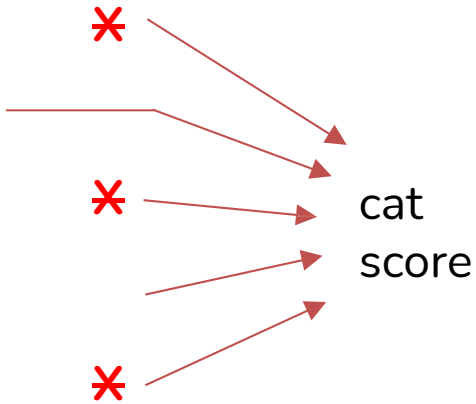
A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron, leading to over-fitting of the training data.

How could this possibly be a good idea?

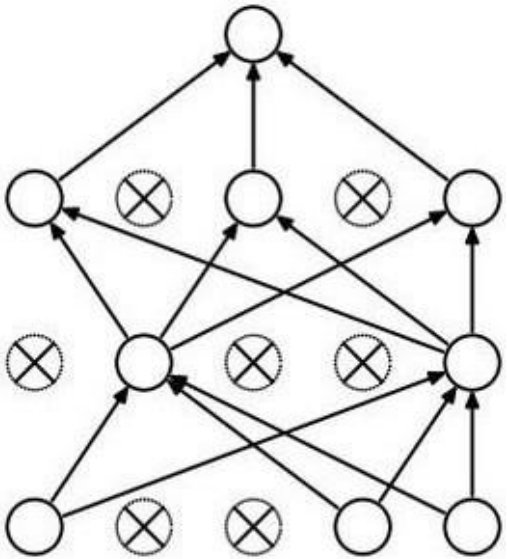


Forces the network to have a redundant representation.





How could this possibly be a good idea?



Another interpretation:

Dropout is an approach to regularization in neural networks which helps reducing interdependent learning amongst the neurons.

At test time...

At test time all neurons are always **ON**

We must scale the activations so that for each neuron:

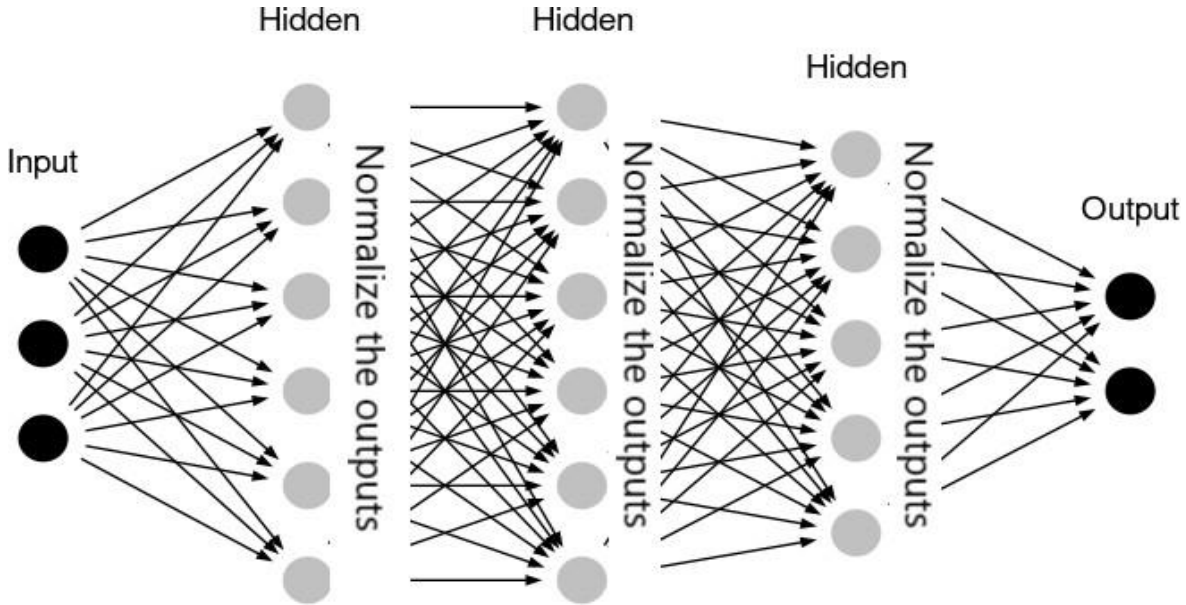
output at test time = expected output at training time

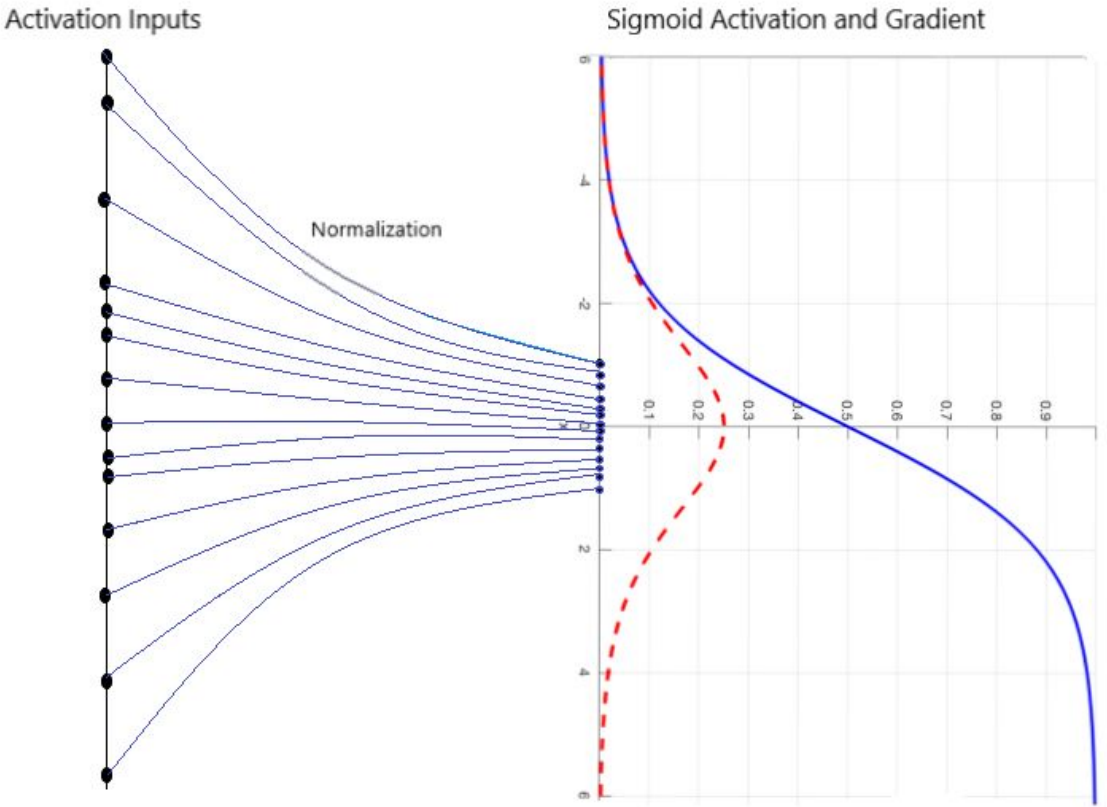
Batch Normalization

Batch normalization is a technique for improving the performance and stability of neural networks.

The idea is to normalise the inputs of each layer in such a way that they have a mean output activation of zero and standard deviation of one.

This is analogous to how the inputs to networks are standardised.





Benefits of Batch Normalization

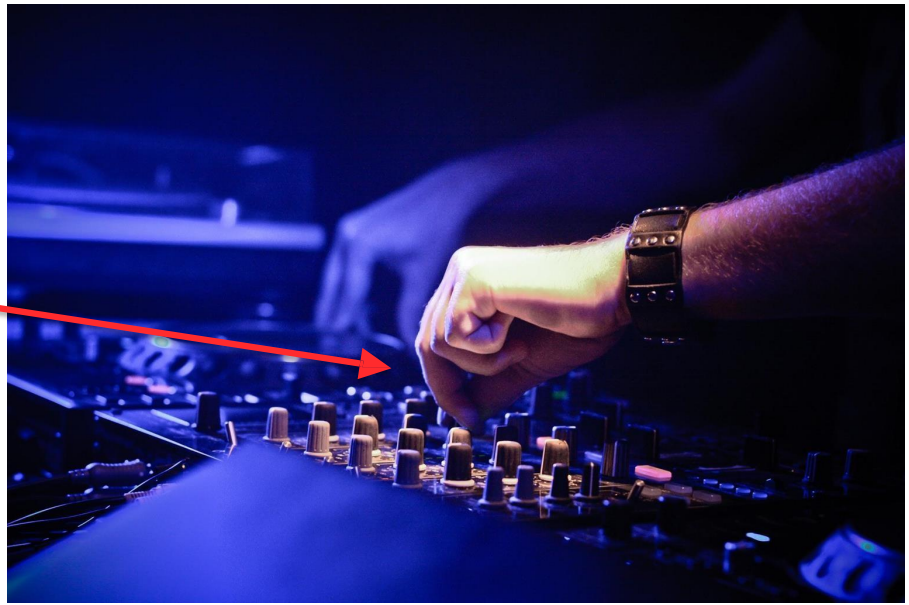
1. Network trains faster
2. Provides some regularization

Hyperparameter Optimization

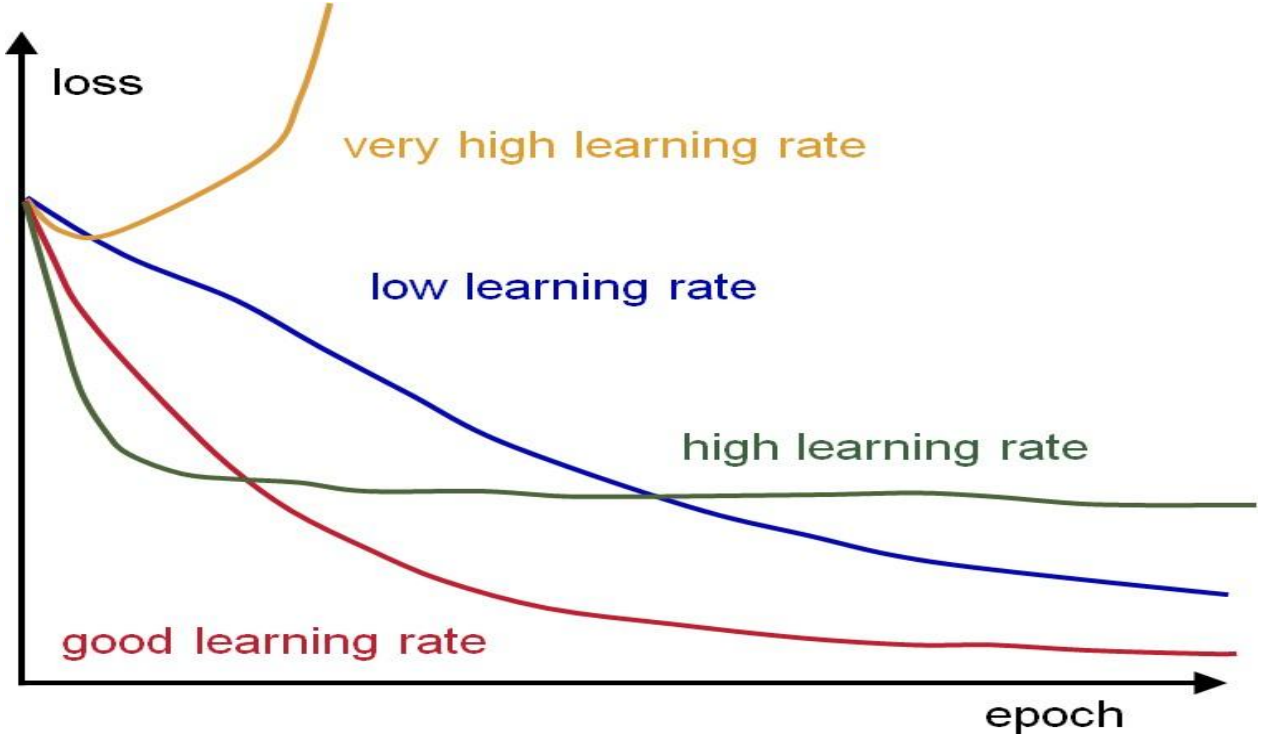
Hyperparameters to play with

- Network architecture
- Learning rate, its multiplier schedule
- Regularization (L2/Dropout strength)

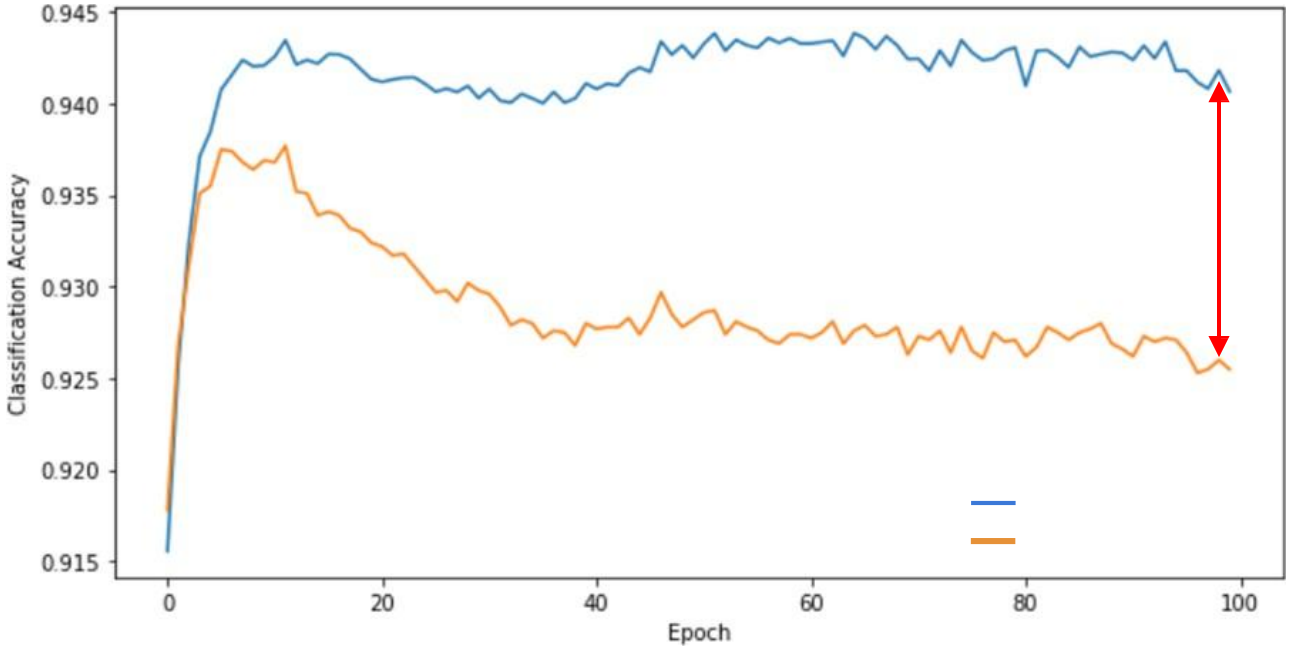
Neural networks practitioner
music = loss function



Monitor and visualize the Loss Curve



Monitor and visualize the Loss Curve



big gap = overfitting

=> increase regularization strength?

no gap - low training and validation accuracy

=> increase model capacity?

Recipe for training an ANN

1. Become one with the data
2. Set up the end-to-end training/evaluation skeleton + get dumb baselines
3. Overfit
4. Regularize
5. Tune
6. Squeeze out the juice

Please refer to this article from Andrej Karpathy

greatlearning
Power Ahead

Happy Learning !

Appendix

Batch Normalization

Due to these normalization “layers” between the fully connected layers, the range of input distribution of each layer stays the same, no matter the changes in the previous layer.

Given x inputs from the k-th neuron. Consider a batch of activations at some layer. For making each feature dimension unit gaussian,

Use:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Scale and Shift

There are usually two stages in which Batch Normalization is applied:

1. Before the activation function (non-linearity)
2. After non-linearity

For sigmoid and tanh activation, the normalized region is more linear than nonlinear.

A few issues:

For sigmoid and tanh activation, the normalized region is more linear than nonlinear.

For ReLU activation, half of the inputs are zeroed out.

So, some transformation has to be done to move the distribution away from 0.

A **scaling factor γ** and **shifting factor β** are used to do this.

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}$$

Final Definition of Batch Normalization

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Where

$\gamma^{(k)}$ and $\beta^{(k)}$ are parameters learnable

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

Batch Normalization - Mini Batches

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$