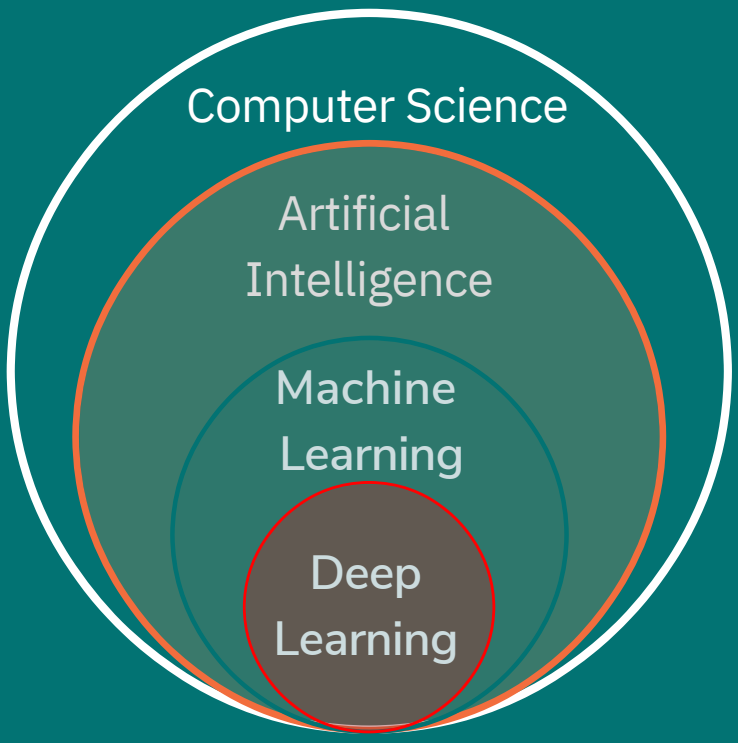
FUNDAMENTALS OF DEEP LEARNING

Meticulously curated information on the base concepts of deep learning presented in diagrams and visualizations.

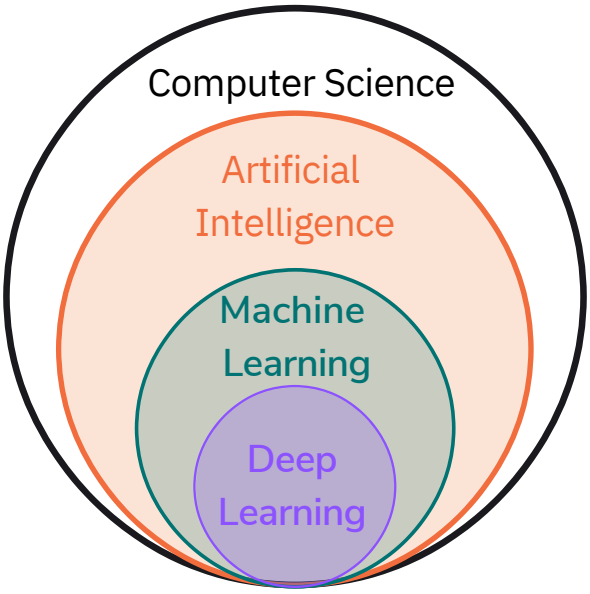


Mısra Turp

What is deep learning?

Deep learning is a discipline of Artificial Intelligence that is based on Neural Networks. It has the power to learn complex patterns directly from the data.

- Deep Learning is a branch of Machine Learning.
- Machine Learning is one of the approaches to Artificial Intelligence.
- Artificial Intelligence is a research branch of Computer Science.



Comparison of Traditional Machine Learning and Deep Learning

	Traditional Machine Learning	Deep Learning
Feature extraction	Needed	Not Needed
Computational Power	Laptop would work fine	High comp. power needed
Amount of Data Needed	Small datasets are fine	Big datasets needed

Most Common Deep Learning Techniques

Deep Neural Networks

Not very complex problems, structured data

Convolutional NNs

Works well with image data, used in computer vision

Recurrent NNs

Works well with sequential data such as language, audio

Generative Adversarial Networks

Can generate real-looking image, video or audio

Reinforcement Learning

Playing games, robotics

Autoencoders

Learns representation of data feature/anomaly detection

Transformers

Used a lot in NLP, machine translation

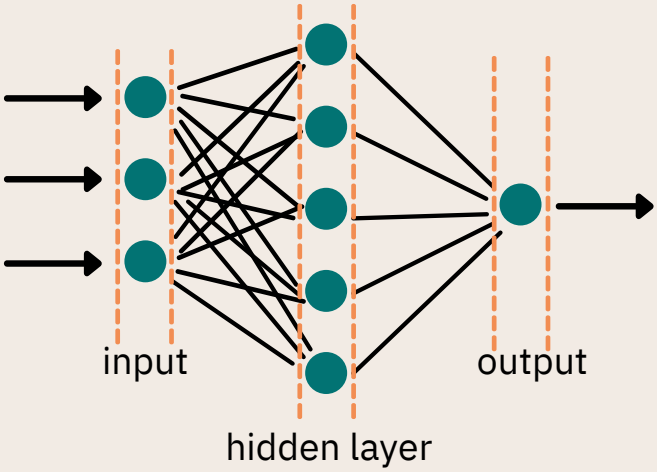
Deep Boltzman Machine

Object/speech recognition

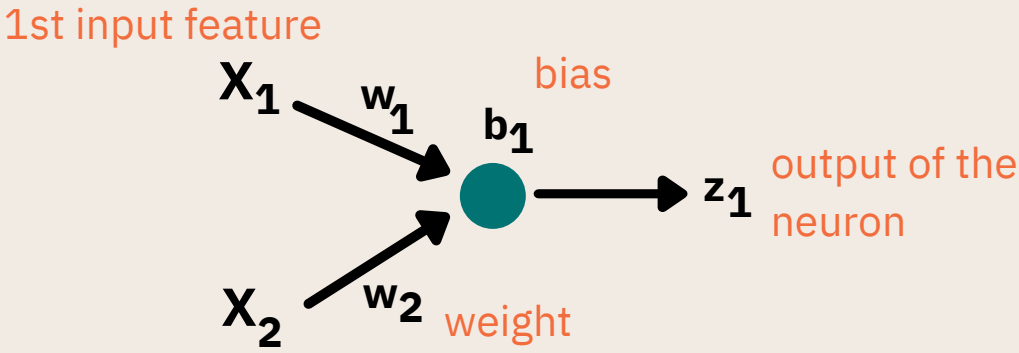
Deep Belief Networks

recognize and generate images, video sequences

Types of layers



A single neuron



Calculating the output of a single neuron

without the activation function

$$z_1 = (w_1x_1 + w_2x_2) + b_1$$

with the activation function

$$a = \alpha\left((w_1x_1 + w_2x_2) + b_1\right)$$

activation function

Vectorization for quick calculations

equation to calculate

weights of the network

biases of the network

inputs

$$A = \alpha\left(W^T X + b\right) \quad W^T = \begin{bmatrix} w_1 w_2 \\ w_3 w_4 \\ w_5 w_6 \\ w_7 w_8 \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} w_1 w_2 \\ w_3 w_4 \\ w_5 w_6 \\ w_7 w_8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_1 x_1 + w_2 x_2 \\ w_3 x_1 + w_4 x_2 \\ w_5 x_1 + w_6 x_2 \\ w_7 x_1 + w_8 x_2 \end{bmatrix}$$

$$\begin{bmatrix} w_1 x_1 + w_2 x_2 \\ w_3 x_1 + w_4 x_2 \\ w_5 x_1 + w_6 x_2 \\ w_7 x_1 + w_8 x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} w_1 x_1 + w_2 x_2 + b_1 \\ w_3 x_1 + w_4 x_2 + b_2 \\ w_5 x_1 + w_6 x_2 + b_3 \\ w_7 x_1 + w_8 x_2 + b_4 \end{bmatrix}$$

$$\alpha\left(\begin{bmatrix} w_1 x_1 + w_2 x_2 + b_1 \\ w_3 x_1 + w_4 x_2 + b_2 \\ w_5 x_1 + w_6 x_2 + b_3 \\ w_7 x_1 + w_8 x_2 + b_4 \end{bmatrix}\right) = \begin{bmatrix} \alpha\left(\left(w_1 x_1 + w_2 x_2\right) + b_1\right) \\ \alpha\left(\left(w_3 x_1 + w_4 x_2\right) + b_2\right) \\ \alpha\left(\left(w_5 x_1 + w_6 x_2\right) + b_3\right) \\ \alpha\left(\left(w_7 x_1 + w_8 x_2\right) + b_4\right) \end{bmatrix}$$

Activation functions

What is an activation function?

An activation function is a transformation on the output of a neuron.

Each layer (except input layer) has it's own activation function.

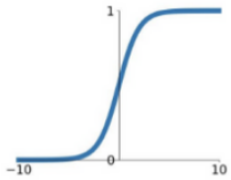
There are many different types of activation functions.

Some examples are:

- Linear
- Sigmoid
- Tanh
- ReLU

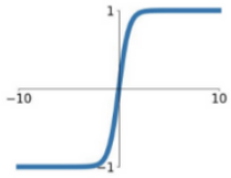
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



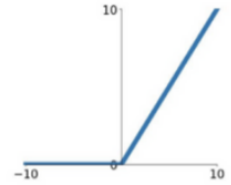
tanh

$$\tanh(x)$$



ReLU

$$\max(0, x)$$



Why do we need a non-linear activation function?

A non-linear activation function is what makes the network learn complex patterns that need something more than linear functions to be represented.

If linear functions are used in the hidden layer, the network can only do linear transformations of the outputs. And no matter how big the network gets it will be as good as a single neuron because of this linearity. This will cause the network to not be able to fit complex problems.

Activation function rules of thumb

For hidden layers:

- Hidden layer activation should never be linear.

For output layer:

- Sigmoid for binary classification and multi-label classification.
- Softmax for multi-class classification.
- ReLU if you need strictly positive outputs
- Linear activation for regression.

Backpropagation

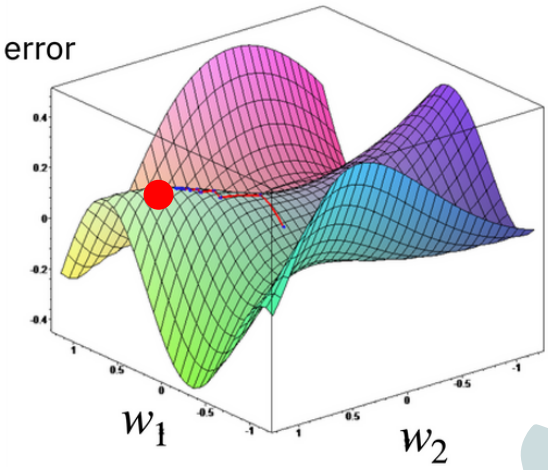
Backpropagation pseudo algotihm

- Initialize the network with randomly generated weights
- Do forward propagation step and calculate the output
- Calculate the error/loss/cost
- Find out how much each parameter contributes to the error
- Calculate this ratio using all the data points in this batch

At the end what you have is a list of ratios of how each parameter effects the error. It is called the gradient vector.

Gradient vector =

$$\begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial w_2} \\ \cdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$



You can think of it as a many-dimensional graph that shows the relationship between each parameter and the cost. If we had only two parameters, the graph would look like this.

Gradient Descent

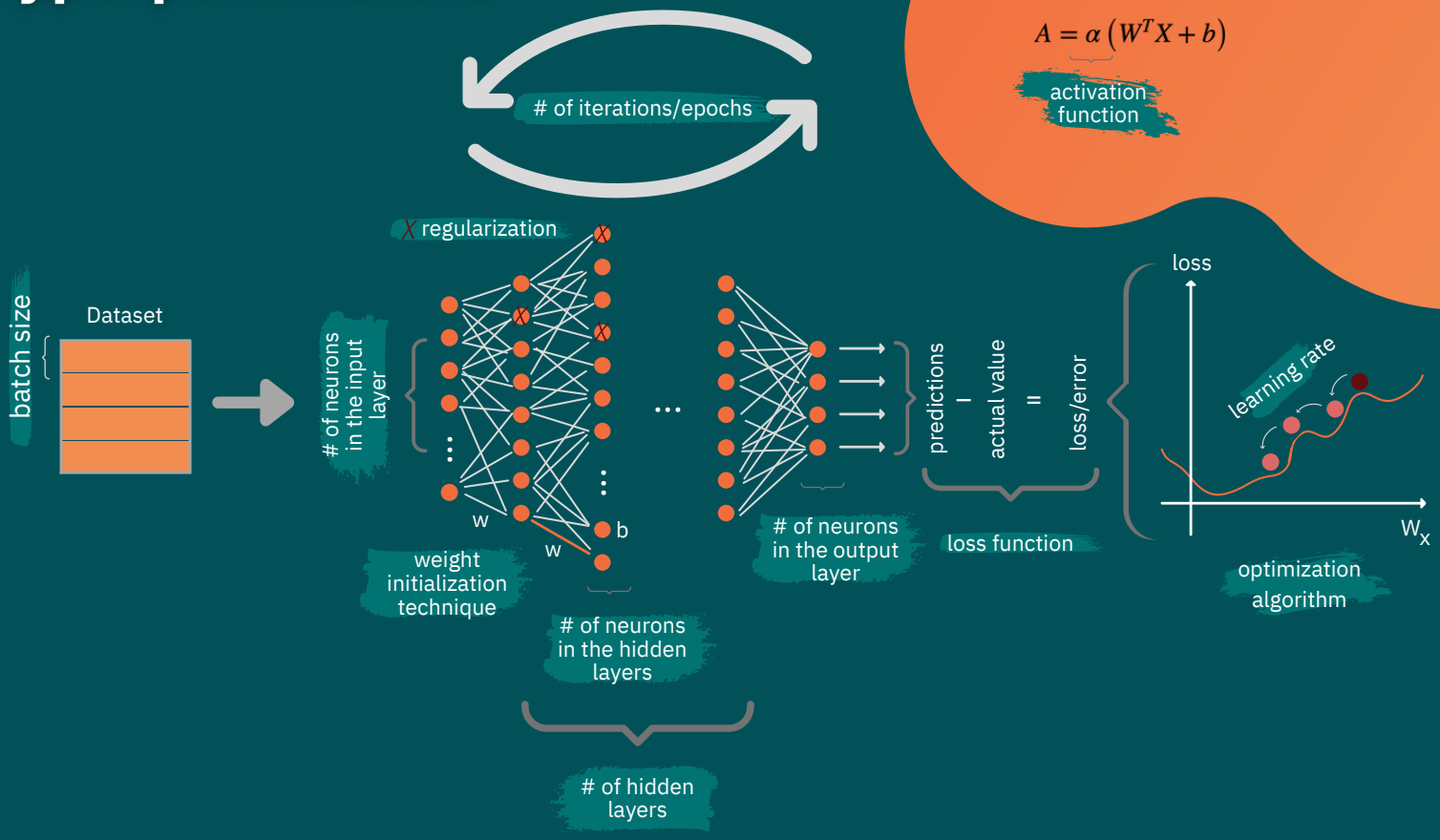
Now that we have this many-dimentional graph showing us how the relationship between all the parameters and the cost works we can decide in what way to update these parameters to lower the cost.

Because the graph would be many many-dimentional, in real-life we cannot actually look at it and decide where to go. **That's why we use the gradient vector.**

Gradient descent is the act of changing the parameters based on the gradient vector. Here is how it works.

new values for parameters = **old values of parameters** − **learning rate** X **gradient vector**

Neural Networks Hyperparameters



Pre-determined hyperparameters

Number of neurons in the input layer

This will depend on the data you are training with.

If you have n features in your dataset you will have n input neurons. If your input is a photo with nxn pixels, you will have n^2 input neurons.

Number of neurons in the output layer

This will again depend on the data you are training with.

If you are doing binary classification or regression, 1 output neuron will do. If you are doing multi-class or multi-label classification, you need as many output neurons as classes/labels.

Hyperparameters that need tuning

Number of hidden layers

The more hidden layers you have, the deeper your network is going to be. Generally, having more layers will help your network more than increasing the number of neurons in the hidden layers.

Number of neurons in the hidden layers

Each hidden layer can have a different number of neurons. This value needs to be adjusted based on how the network is performing.

Activation function

Each layer has its own activation function except the input layer. The activation function of the hidden layers cannot be linear.

Optimization algorithm

The optimization algorithm determines how the network updates its weights.

Loss function

The loss (cost) function is what we're trying to minimize during training. The cost function you use will depend on the type of problem you have.

Batch size

Batch size is the amount of data points you put in each batch while training your network.

If you use a batch size of 1, you will be doing **Stochastic gradient descent (GD)**. If you use batch size equals the number of data points in your whole dataset, that is **Batch GD**. If batch size is anything between 2 to number of data points in your whole dataset, you are doing **mini-batch gradient-descent.**

Number of epochs

This is how many times you run the whole dataset through your network. The network learns the dataset a little bit better after each epoch.

Weight initialization technique

This is how the weights of the networks are initialized. Bias values can be (and often are) initialized to zero. But weights cannot be initialized to zero because if they are, they will all be updated the same way and the model won't be able to learn.

At the same time, just using normal distribution for the random initialization of the weights cause problems with the network. That's why, sometimes we need a more advanced approach.

Learning rate

Learning rate determines how big of a step to take towards the direction set by gradient descent. Too small of a learning rate might slow down learning and too big of a learning rate might cause the network to keep missing the optima.

Learning rate scheduling techniques are used to dynamically decide on the learning rate value.

Regularization

Regularization is a way to deal with overfitting. There are many different ways to do regularization and often these techniques come with their own hyperparameters.

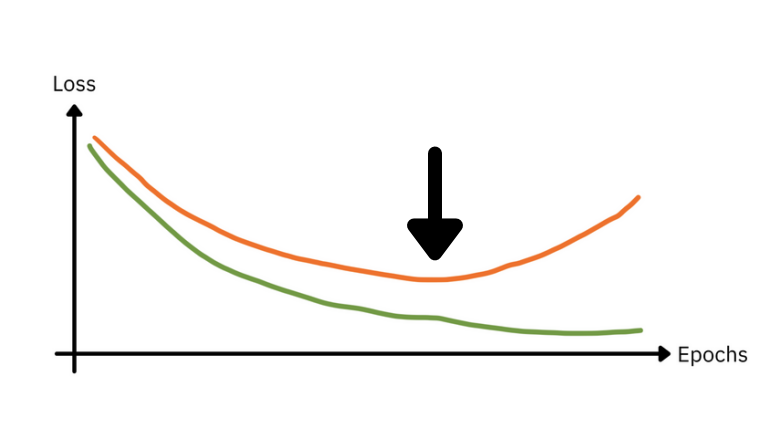
Some examples are:

- L1
- L2
- Dropout

L1 and L2 regularization has a hp called alpha and dropout has a hp called the dropout rate.

What is overfitting?

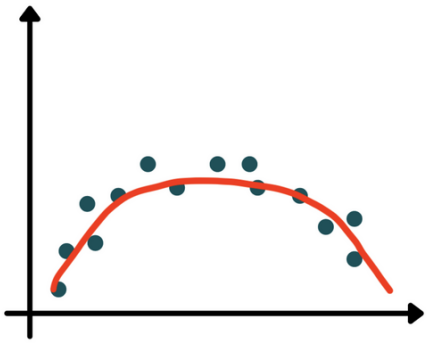
Overfitting is when the model is not able to generalize well. The model performs well on the training set but fails to capture the same performance for the validation set. We see this in the comparison of training and validation loss. As we train the network more, the training loss keeps getting lower whereas after a point validation loss starts increasing.



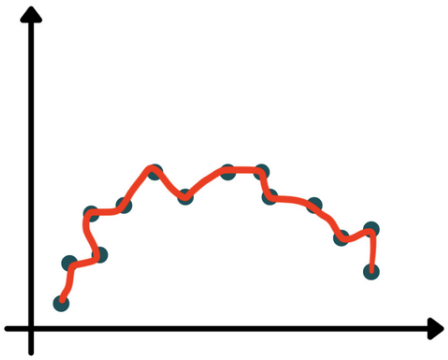
What is underfitting?

Underfitting is when the model is not able to fit the data at all. We see this when the performance of the network is very low already on the training set.

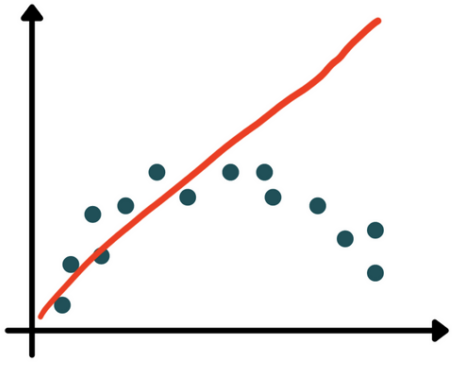
If this is the ideal way to fit a dataset:



This is what it would look like when overfitted:



Or when underfitted:



Bias and Variance

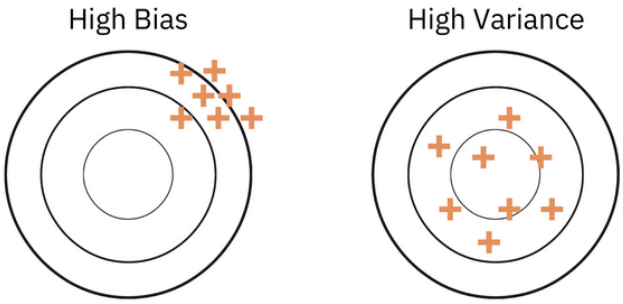
Bias: The amount of assumptions a model makes about the data. The more assumptions it has, the simpler the model will be.

Variance: dependence of the model on the particular training set that was used to train it.

If a model has **overfit** it means its **bias is low and variance is high.**

If a model has **underfit** it means its **bias is high and variance is low.**

If the mid point of these circles signify where all the correct values lie, the predictions of high bias and high variance models will look like the yellow marks.



Solutions to high bias/variance

Problem	High Bias (Training performance is low)	High Variance (Validation performance is low)
Causes	Underfitting	Overfitting
Solutions	• Train more • Increase model complexity • Try a different model architecture	• Introduce more data • Use regularization • Try a different model architecture

Regularization

Constraining a model to simplify it

- L1 / L2 regularization
- Drop out
- Early stopping

Adding more information

- Data augmentation

L1 / L2 regularization

Works by lowering the weights of the network. Achieves this by adding the weight values to the cost function.

L1 regularization: Add the sum of the absolute values of the weights to the loss.

L2 regularization: Add the sum of the squared values of the weights to the loss.

The alpha parameter: how much attention to pay to this addition to the cost function. Value between 0 (no penalty) and 1 (full penalty).

Tips

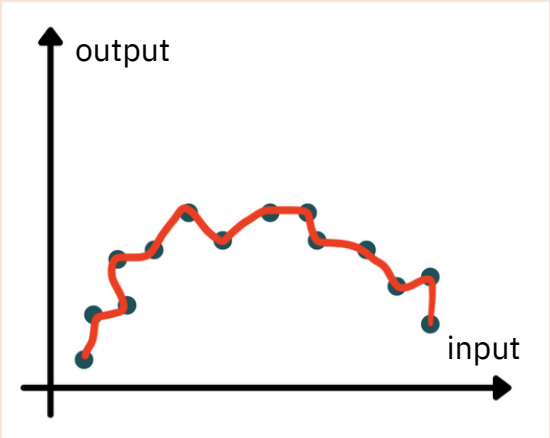
- You can use them with all network types
- Normalize input for best results
- Use L1+L2 together

Note!

L1 and L2 regularization lowers the weights of the network to combat overfitting but **what does overfitting have to do with high weights?**

When you have high weights, that means you are exaggerating the importance of a certain input. When you overfit this is what the model looks like, right?

It looks like the importance of this input is so exaggerated that the model follows its pattern to the full.



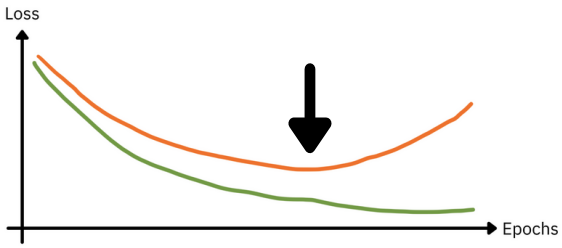
Dropout regularization

Works by making some neurons inactive in every training step. Every neuron has a probability ***p*** of being inactive. This is called the dropout rate.

In training time, on average ***1/p*** neurons are inactive. During test time all neurons are active. That's why the input to each neuron is multiplied with the keep probability which is ***(1-p)***.

Early-stopping

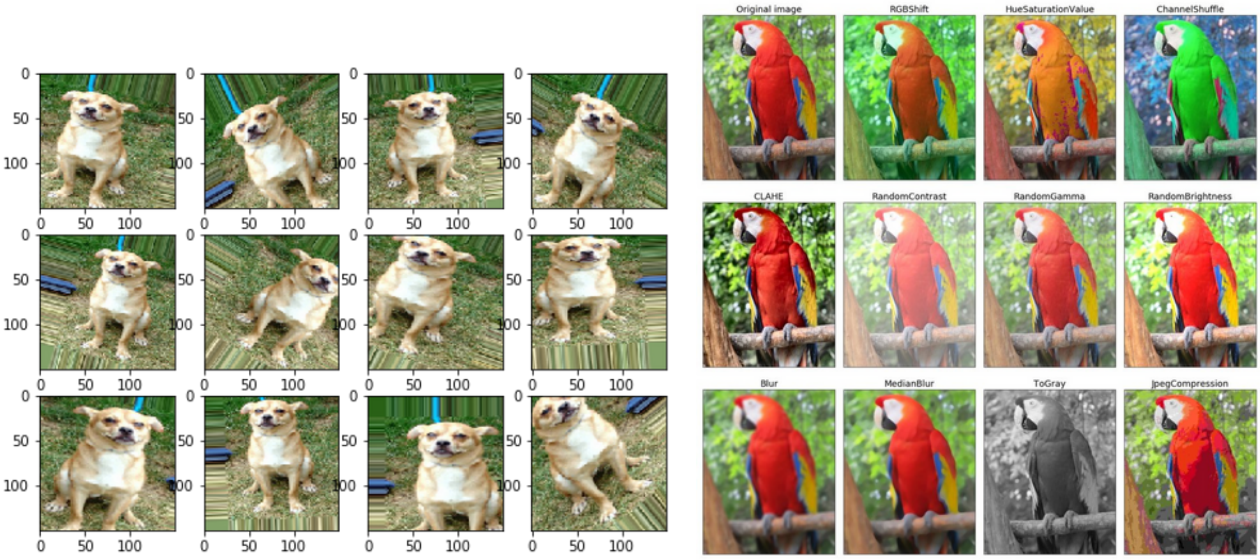
Works by stopping training at the point the validation loss starts getting higher. This is the same place we mentioned before where overfitting starts happening.



But early-stopping is not the best approach to use for overfitting. Because training and mitigating overfitting should be separate processes where separate approaches and techniques are used. Early-stopping combines these two.

Data augmentation

Transforming your data in multiple ways before feeding it to your network. This way, you can generate more data points from the same one. These transformations help the model tolerate any of the changes made (e.g. flipping, different orientation, color changes, RGB or black and white).



Images from http://datahacker.rs/tf-data-augmentation/ and Buslaev, Alexander & Parinov, Alex & Khvedchenya, Eugene & Iglovikov, Vladimir & Kalinin, Alexandr. (2018). Albumentations: fast and flexible image augmentations.

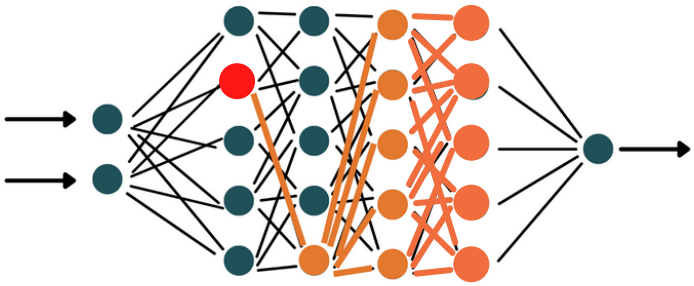
Unstable gradients

Gradients are what we use to update the parameters of the network.

Sometimes they get extremely small or extremely big.

This is because while calculating the gradient of a parameter (that is how much this parameter effects the cost), we need to multiply a lot of values from other neurons.

In the example of the right, to find out the gradient of the parameters of the red neuron, we need to work our way all the way to the output neuron.



In this calculation a lot of small numbers are multiplied with each other, causing a very small number for the gradient to be calculated.

small number X small number X ... X small number = very small number

The reverse could happen if the parameter values happened to be big values. That way we would get the

Solutions

Changing weight initializers

The way of initializing the weights might contribute to the unstable gradients problem. There is a need for a strategy that is better than just initializing them with normal distribution and a mean of zero. Here are some of the initializers:

Glorot (Xavier) Initialization

$$\text{Mean} = 0$$

$$\sigma^2 = \frac{1}{fan_{avg}}$$

$$fan_{avg} = \frac{fan_{in} + fan_{out}}{2}$$

He Initialization

$$\text{Mean} = 0$$

$$\sigma^2 = \frac{2}{fan_{in}}$$

LeCun Initialization

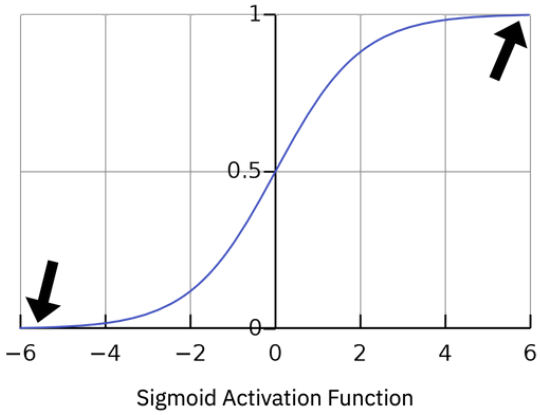
$$\text{Mean} = 0$$

$$\sigma^2 = \frac{1}{fan_{in}}$$

Where fan_{in} is the number of inputs and fan_{out} is the number of outputs of a neuron.

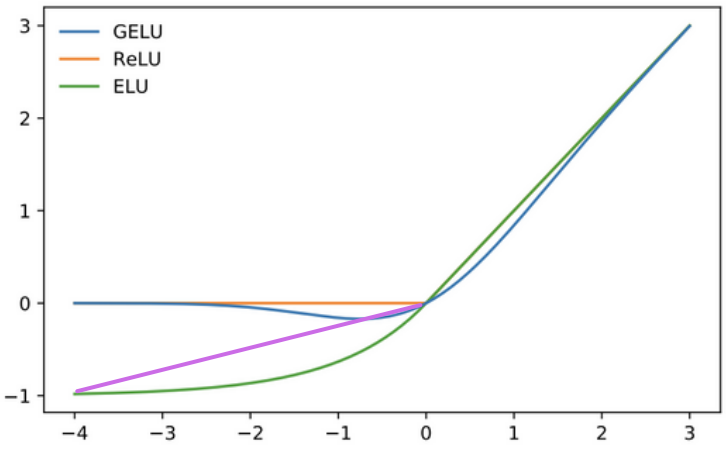
Using a non-saturating activation function

Some activation functions cause saturation on the extremes like seen in the sigmoid function to the right. In combination with using the wrong initialization technique, this causes the unstable gradients problem.



Sigmoid Activation Function

ReLU (Rectified Linear Unit) activation function does not cause saturation for positive values. But it still causes saturation for negative values. So there are a bunch of variations on it.



- Rectified Linear Unit (ReLU)
- Leaky ReLU (with parameter alpha)
- Randomized Leaky ReLU (RReLU)
- Parametric Leaky ReLU (PReLU)
- Exponential Linear Unit (ELU) (with parameter alpha)
- Scaled ELU
- Gaussian Error Linear Units (GELU)

Which activation function to use when?

Try the activation functions in this order:

- SELU
- ELU
- Leaky ReLU (and/or variants)
- ReLU
- Tanh
- Logistic (sigmoid)

If speed is a priority ReLU is a good choice because most libraries adapted fast ReLU implementations.

You should choose the correct initializer for your activation function

Initializer	Activation function
Glorot	Linear, tanh, softmax
He	ReLU and variants of ReLU
LeCun	SELU

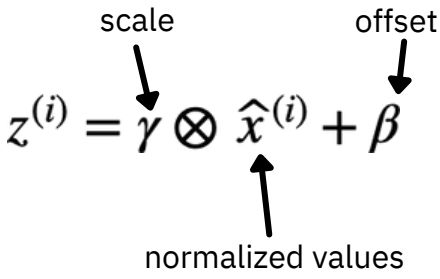
To use SELU, make sure:

- your input is standardized (mean=0, std=1),
- you are using LeCun initializer,
- that you have a sequential network.

Batch normalization

Centers the inputs to all layers at zero and sets standard deviation of them to 1. After that scales and offsets them by 2 trainable values.

Scale and offset values that are the best for this network, is learned during training.



Advantages of batch normalization

1. Even after setting a good activation function and initializer unstable gradients might occur. Batch normalization stops the possibility of unstable gradients.
2. Eliminates the need for manually adding a standardization layer.
3. Makes the network converge to the optimum faster.
4. Reduces the need for regularization

Gradient clipping

Batch normalization is very tricky to be used on RNNs. That's why instead gradient clipping is used to deal with exploding gradients.

Gradient clipping is, like the name suggests, clipping the values of the gradients as they get too big.

$$\text{Gradient vector} = \begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix} \quad \text{Gradient clipping range} = [-1.1]$$

Clipping by absolute values

$$\begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix} \rightarrow \begin{bmatrix} 0.9 \\ 1.0 \\ 1.0 \\ -1.0 \\ 0.3 \end{bmatrix}$$

Clipping by norm

$$\begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix} \rightarrow \begin{bmatrix} 0.006 \\ 0.021 \\ 1.0 \\ -0.014 \\ 0.002 \end{bmatrix}$$

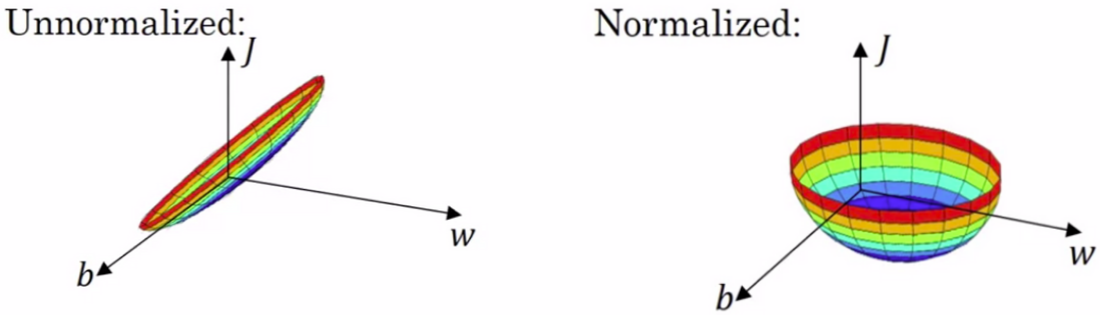
Techniques to speed up training

- Applying **good initialization**
- Using a **good activation function**
- Using **batch normalization**
- Reusing parts of a **pretained network**
- **Normalizing** the data
- Using **mini-batches**
- **Learning rate scheduling**
- A **faster optimization algorithm**
- **Pruning** the network

Normalizing the data

When you train your network with unnormalized data, the parameters of the network will have a more complicated relationship with the cost function.

Specifically, the graph of relationship between the parameters and the cost will look like a very wide bowl. The edges of this bowl is steep, so gradient descent will make quick progress at first but as we get closer to the minima (which is the point where the cost is the lowest) the progress will get smaller and smaller because the slope of the graph is very small.

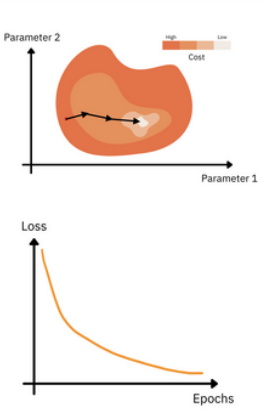
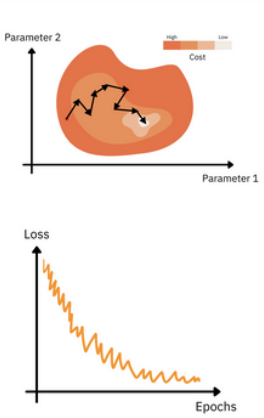
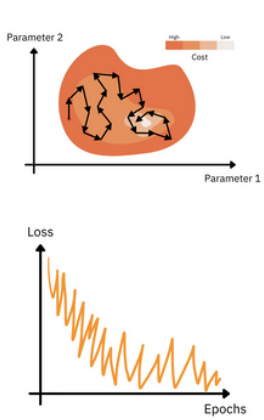


Whereas with normalized data, the relationship graph will look more like a smooth bowl. Thus, the progress will be faster since the direction of the minima is very clear from anywhere on the bowl.

Images from www.coursera.org/learn/deep-neural-network/

Using mini-batches

Batch Gradient Descent (GD): Run the whole data through the network
Mini-batch GD: Run 2 or more but less than the whole data through the network
Stochastic GD: Run examples one by one

	Batch GD	Mini-batch GD	Stochastic GD
Learning pattern			

	Batch GD	Mini-batch GD	Stochastic GD
Advantages	• Less oscillations towards the minima	• Generalizes well • Fast	• Fast to calculate output • Can get out of local minima
Disadvantages	• One iteration takes very long	• Another parameter to tune	• Loses speed gained from vectorization • Will not converge

Batching rules of thumb

- Small dataset (<2000 data points) use Batch GD
- Otherwise use mini-batch GD

- Use mini-batch sizes that are powers of 2.
- Research suggests you can use as much as 8192*.
- 2 to 512 is typical.
- Find a number that fits in your CPU/GPU.
- Start big and lower if performance is bad.

*Elad Hoffer et al. and Priya Goyal et al. Given that you use learning rate warming up.

Using a faster optimization algorithm

Gradient Descent

$$\text{Change} = \underbrace{\eta}_{\text{learning rate}} \underbrace{\begin{bmatrix} \frac{\partial C}{\partial \theta_1} \\ \frac{\partial C}{\partial \theta_2} \end{bmatrix}}_{\text{gradient vector}}$$

$$\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \text{change}$$

Gradient Descent with Momentum

$$\underbrace{m}_{\text{Change}} = \underbrace{\beta\, m}_{\text{change from previous step}} - \underbrace{\eta}_{\text{learning rate}} \underbrace{\begin{bmatrix} \frac{\partial C}{\partial \theta_1} \\ \frac{\partial C}{\partial \theta_2} \end{bmatrix}}_{\text{gradient vector}}$$

$$\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + m$$

Nesterov Accelerated Gradient

$$\underbrace{m}_{\text{Change}} = \underbrace{\beta\, m}_{\text{change from previous step}} - \underbrace{\eta}_{\text{learning rate}} \underbrace{\begin{bmatrix} \frac{\partial C}{\partial(\theta_1 + \beta m)} \\ \frac{\partial C}{\partial(\theta_2 + \beta m)} \end{bmatrix}}_{\text{gradient vector at a future point}}$$

$$\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + m$$

Other algorithms

- AdaGrad
- RMSProp
- Adam
- Nadam

Here is a comparison of optimizers based on Aurélien Géron's book*

Optimization algorithm	Convergence speed	Convergence quality
Gradient descent (GD)	Bad	Good
GD with momentum	Average	Good
GD with momentum and Nesterov	Average	Good
AdaGrad	Good	Bad (stops too early)
RMSProp	Good	Average to Good
Adam	Good	Average to Good
Nadam	Good	Average to Good
Adamax	Good	Average to Good

*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems

Pruning the network

Pruning the network gets rid of parameters with very small values, creating a sparse network. This network will take up less space in memory and the prediction times of the network will be faster. Ways to prune a network is:

- **Using L1 regularization** at training time
- **Removing very small weights manually** (will probably lead to worse performance)
- **Use tools** like TensorFlow Model Optimization Toolkit

Learning rate scheduling

Dynamically determining what the learning should be, in order to overcome the limitations of having a pre-determined static learning rate.

Manual approach

Train many models, give them increasing values for the learning rate and choose a value a bit before than when the loss starts shooting up.

Piecewise constant scheduling

Use constant learning rate for a certain number of epochs. Very similar to manual.

$$\eta_0 = 0.1$$
$$\eta_1 = 0.001$$
$$\eta_2 = 0.0001$$

Power scheduling

The learning rate is a function of the epoch number. The rate of decrease, decreases.

$$\eta(t) = \frac{\eta_0}{(1 + t/s)^c}$$

Performance scheduling

Drop the learning rate when the validation error stops dropping.

Can't hurt to try

Recommended

Exponential scheduling

The learning rate is a function of the epoch number. It drops by a factor of 10 every s epochs.

$$\eta(t) = \eta_0 0.1^{t/s}$$

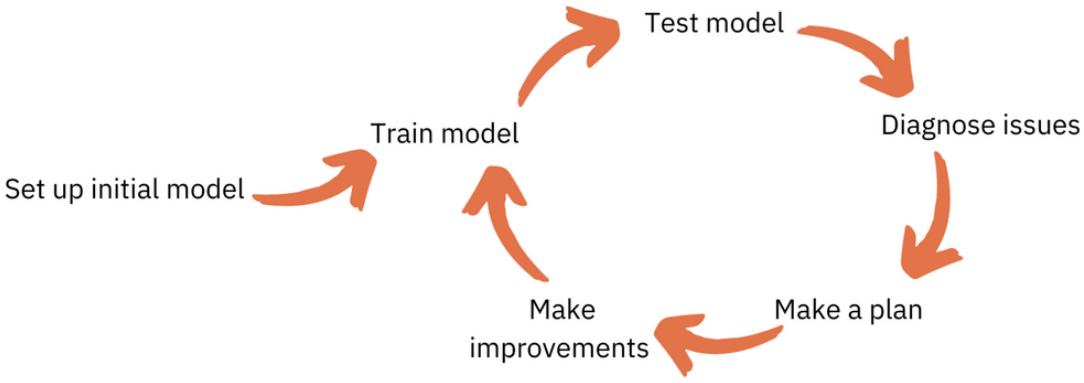
1cycle scheduling

First increase linearly towards n_1 during first half of training. Then lower it back to n_0, during second half of training.

η_0 = ~ 10x lower than η_1

η_1 = Optimal learning rate

Model lifecycle



Evaluating the model

- Have only one metric to compare models
- You can combine multiple metrics to create one metric
  - E.g.: combine precision and recall to have the F-score or average multiple accuracy values
- If necessary, you can add a helper metric
  - E.g.: model speed

Levels of performance

Training set performance

Validation set performance

Test set performance

Real-life performance

The goal of diagnosis is to find out where in these levels the problem occurs and fix it.

How do we know what training set performance is good?

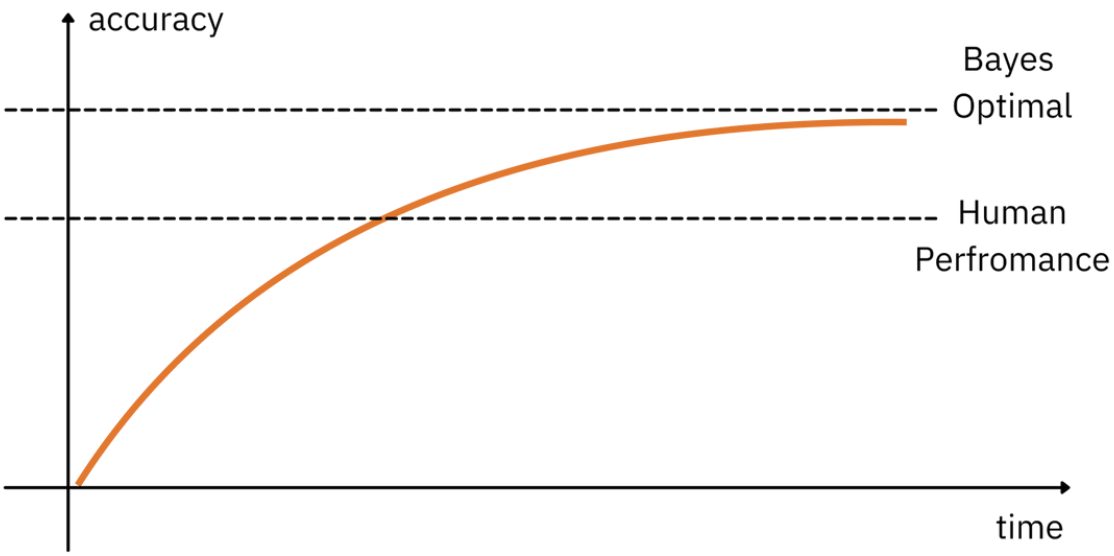
We compare it to human-level performance.

How do we know what real-life performance is good?

We evaluate user satisfaction or whatever other real-life evaluation metrics (this could still be accuracy or precision in real-world tasks).

Bayes Optimal Performance

This is the best way a task can be done. It is a hypothetical limit. Many times we don't know what it is. But most of the time, Bayes Optimal Performance is thought to be the best performance achieved on this task by humans.



Bayes Optimal Performance is always either equal to or better than Human-level performance. Machine learning algorithms (shown in orange line) progress fairly fast to the point of achieving human-level performance. But after they surpass that level, it takes much more effort to make progress towards Bayes Optimal Performance.

Human-level performance

Human-level performance is normally what models aspire to. But it can be defined in many different ways. Let's say we observed these error percentages in a given task by these groups of people:

Inexperienced person	1.5 %
Doctor	0.9 %
Pharmacist	0.5 %
Group of Pharmacists	0.3 %

In this case, the best performance is by Group of Pharmacists. If we have not other information on how well this task can be done, we can accept 0.3% as Bayes Optimal Error.

But what we should accept as human-level performance will depend on our task.

Improving the model

Human-level performance

↕ Bias difference

Performance on training set

↕ Variance difference

Performance on validation set

Deciding which difference is bigger will help us decide what actions to take to make the model better.

That is why, choosing the correct human-level performance for each specific case matters.

As we talked about before, there are things we can do to address high bias or high variance.

Problem	High Bias (Training performance is low)	High Variance (Validation performance is low)
Causes	Underfitting	Overfitting
Solutions	• Train more • Increase model complexity • Try a different model architecture	• Introduce more data • Use regularization • Try a different model architecture

But on top of that, here are what we should do to make the model better when we want to address a specific performance gap.

Human-level performance

- Train more
- Increase model complexity
- Try more advanced architecture
- Find better hyperparameters

Performance on training set

- More data
- Regularization
- Try different architecture

Performance on validation set

- Find better hyperparameters
- Have a bigger validation set

Performance on test set

- Change validation set
- Change cost function

Performance in real-life

Hyperparameter tuning

There are many hyperparameters and many options for hyperparameter in neural networks. That's why, many times, trying out all the combinations of hyperparameter settings is not feasible. Instead, we have some tactics we use to make life easier.

Grid search

Grid search is trying each option one by one. This is might be possible for some traditional machine learning models but it is not really an option for neural networks.

Random search

Random search is trying out a subset of settings in the whole space of possibilities. It does not guarantee finding the best possible combination of hyperparameter settings but it works good enough most of the time.

Manual zooming in

This is a manual way of looking for the best settings. The idea is to do iterative random search. In each iteration, the search is focused around the settings that performed the best in the previous run.

Bayesian search

Build a probabilistic model of the relationship between the hyperparameters and the cost function.

Gradient-based search

Approaches the hyper parameter tuning problem like the learning problem.

Evolutionary computing based search

Uses processes like randomization, natural selection and survival of the fittest to find the best hyperparameter settings.

Early-stopping based search

The main approach of these search algorithms is to focus resources on settings that are promising. There are different types of algorithms (SHA, ASHA, Hyperband) that fall into this cateory.

Which approach to use?

The best way to do hyperparameter tuning, especially for your first couple of projects, is to use random search. Once you have a good grasp on it, you can use different approaches like bayesian search or even evolutionary computing based search.

You do not need to implement these search algorithms yourself. Here are some libraries that can help you:

Bayesian search	Spearmint Scikit-Optimize
Gradient-based search	Adatune
Evolutonary computing based search	Sklearn-Deap
Early-stopping based search	Hyperband
Other libraries and resources	Hyperopt Kopt Talos Hyperas Keras Tuner SHERPA Google CLoud AI Platform HP tuning service SigOpt Oscar