Improving Deep Neural Networks

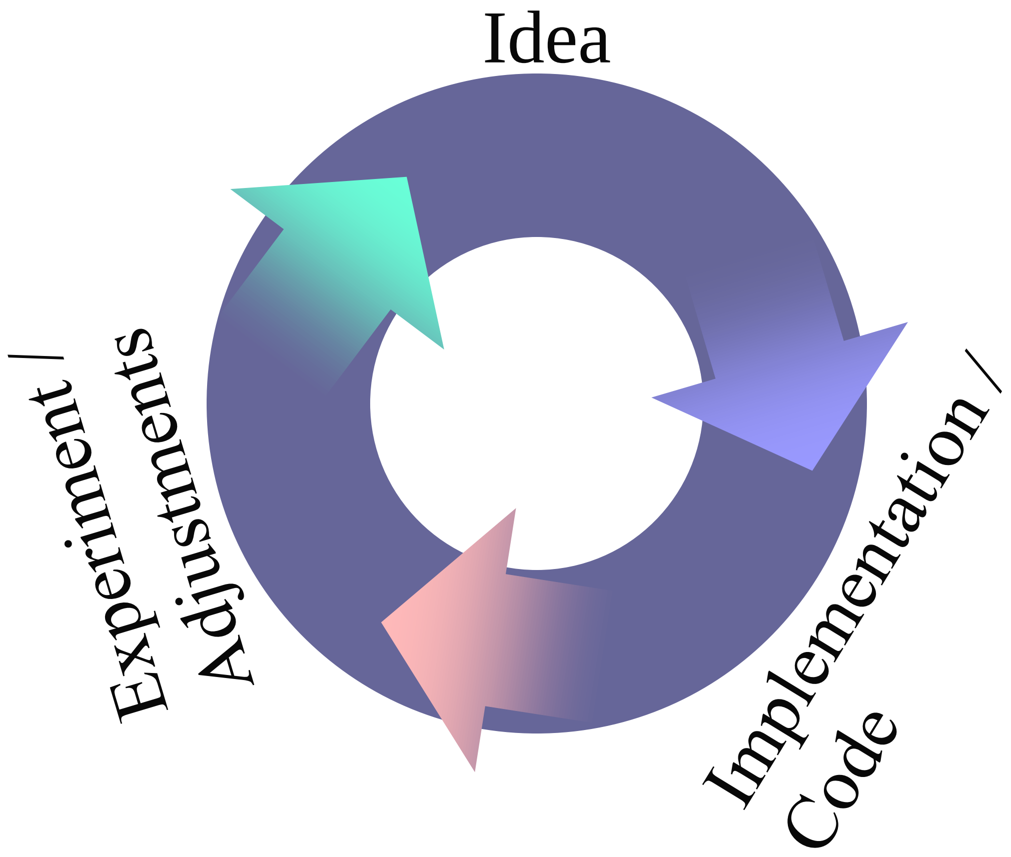
- ➢Data set split
- ➢Regularization
- ➢Hyperparameter tuning
- ➢Optimization

[DeepLearning.AI, Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization, https://www.coursera.org/learn/deep-neural-network/home/welcome]

[SHUBHAM JAIN, An Overview of Regularization Techniques in Deep Learning (with Python code), APRIL 19, 2018, https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/]

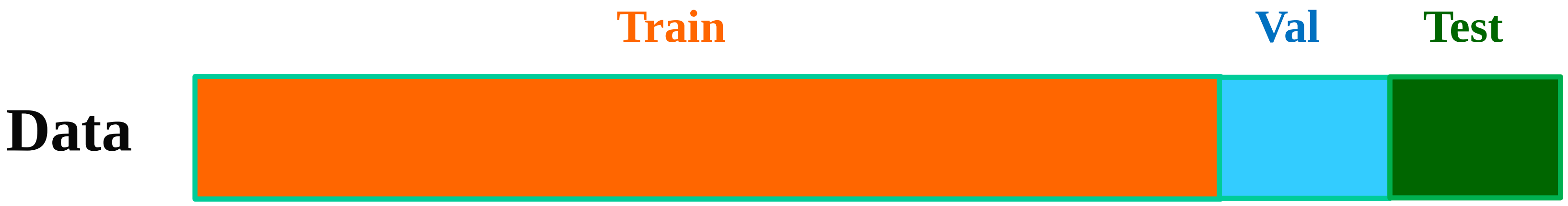
Machine learning – intensive iterative process

Layers
Hidden units
Learning rates
Activation function
Epochs
Batch
…



How efficiently can you go round this cycle?

Train / Val /Test split



Train set: used to learn the parameters of the model

Val set (validation set): supervises the learning generality (identify overfitting);
Used to rank different models in terms of their accuracy (decide which models to proceed further with); parameter choice and model choice

Test set: used as a proxy for unseen data and evaluate our model on test-set (brand-new data set)

Size of training/val/test split

Small / moderate data set:

- **70% / 20% /10%**

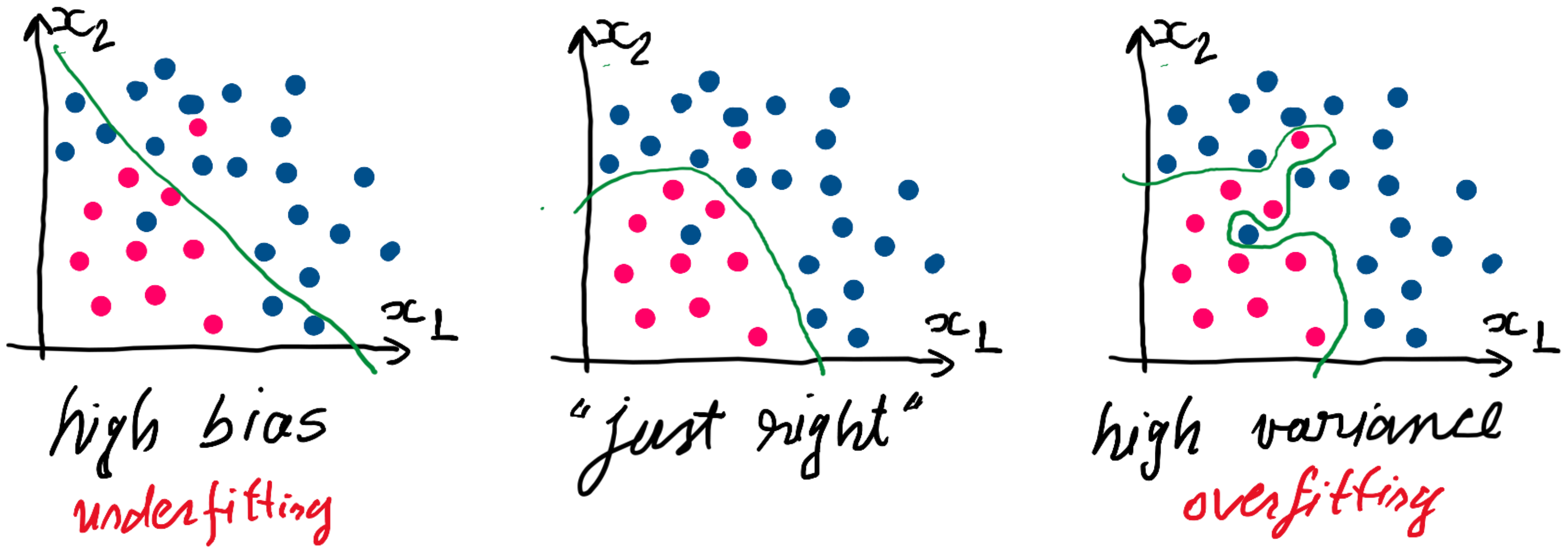
Big data set:

Val set ~ 1000 – 10000 example; Test set ~ 100 – 1000 example

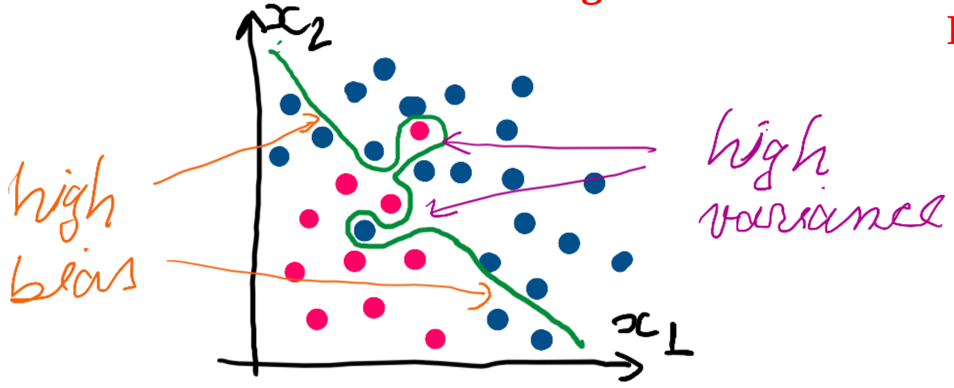
https://snji-khjuria.medium.com/everything-you-need-to-know-about-train-dev-test-split-what-how-and-why-6ca17ea6f35

2D data; binary classifier

Bias / Variance

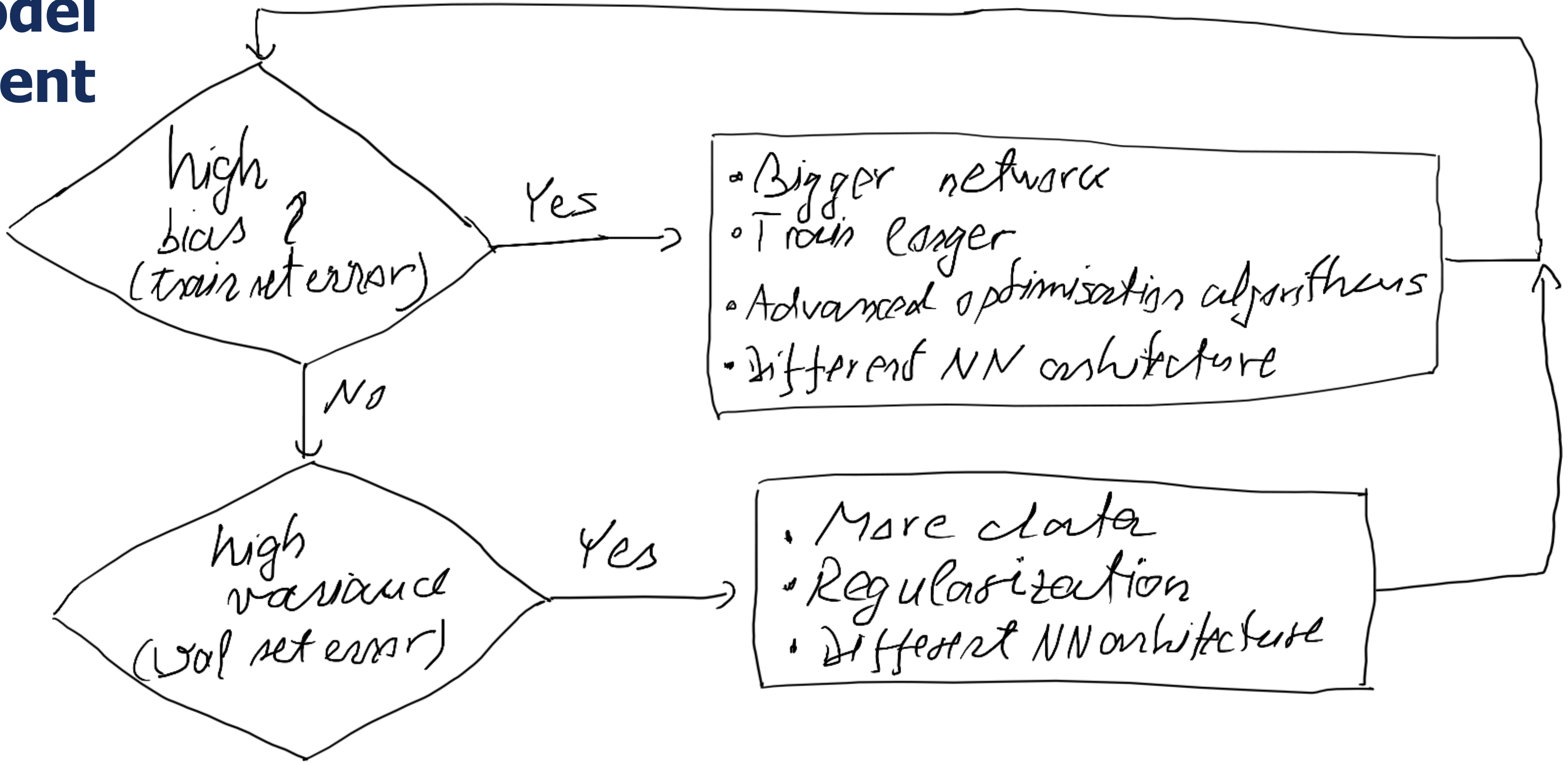


Train set error:	1%	14 %	14%	0.6%
Val set error:	12%	15 %	21%	0.9%
	High variance	Low variance High bias	High variance High bias	Low variance Low bias



Same model can present high bias in one region and high variance in another region !

Basic recipe for ML model development



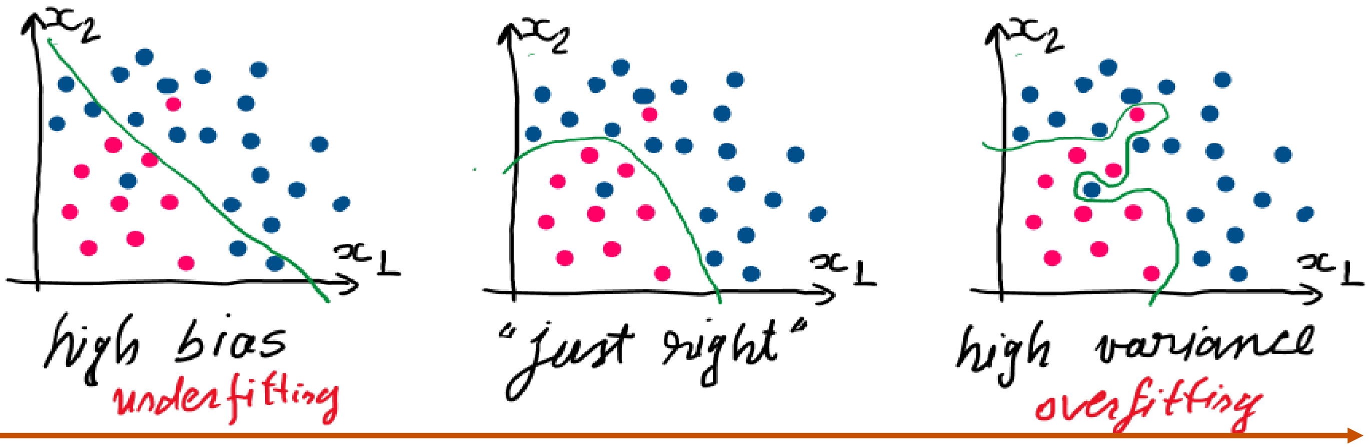
high bias ? (train set error)
Yes
• Bigger network
• Train longer
• Advanced optimisation algorithms
• Different NN architecture
No
high variance (val set error)
Yes
• More data
• Regularization
• Different NN architecture

Bias / variance tradeoff ?
• Not anymore for modern ML

Training a bigger network almost never hurts.

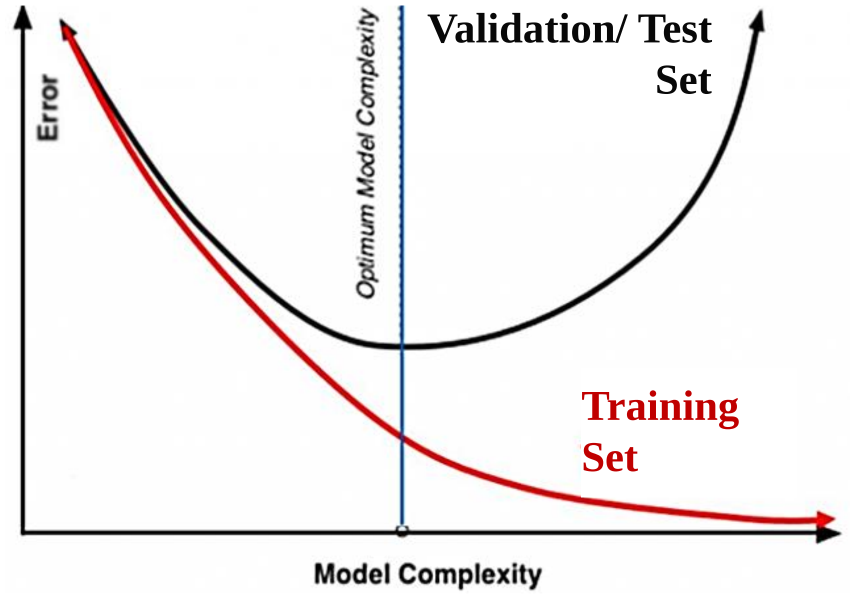
Main cost of training a neural network that's too big is just computational time, so long as you're regularizing (to avoid overfitting).

Overfitting



The model tries to learn too well the details and noise from the training data
Poor performance on the other data (validation/test data)
The complexity of the model increases
Training error decreases   Validation / testing error increases

Training vs. Validation / Test Set Error

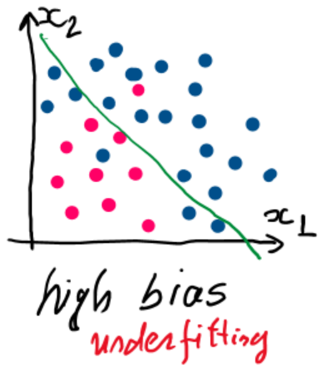


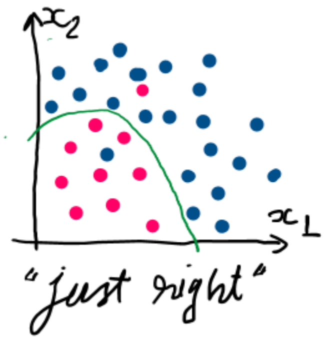
One of the most common problem data science professionals face is avoiding **overfitting**

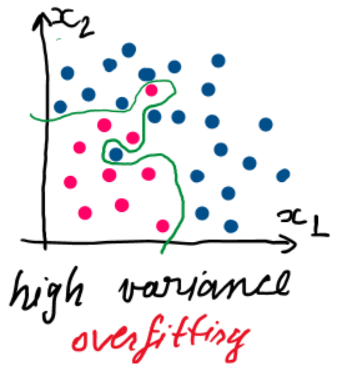
The model

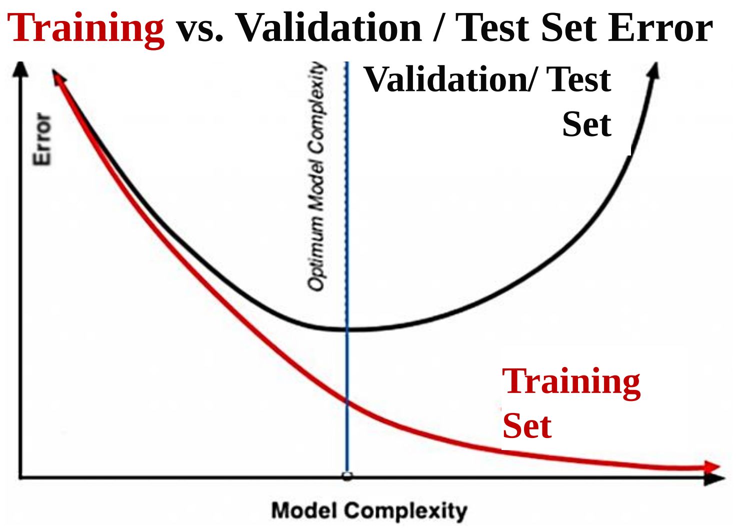
- **exceptionally well on training data,**
- **quite poorly on validation data**
- **not able to make accurate predictions on test data**

Regularization









- ➢ **Avoiding overfitting can single-handedly improve our model's performance**

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 new (unseen) data**

In machine learning, regularization penalizes the coefficients.

In deep learning, it actually **penalizes the weight matrices** of the nodes.

Regularization

- When you decrease the number of training parameters you usually get a lot of benefits such as smaller model making them fit into memory easier.
- However, that usually lowers the performance.
- So, the main challenge is
  - **decrease the number of parameters without lowering the performance**.

A huge regularization effect on small images would cause underfitting and a small regularization effect on large images will not avoid overfitting.

Mostafa Ibrahim, Google releases EfficientNetV2 — a smaller, faster, and better EfficientNet, Apr 3 2021, https://towardsdatascience.com/google-releases-efficientnetv2-a-smaller-faster-and-better-efficientnet-673a77bdd43c

L2 & L1 regularization

If neural network is overfitting the data (high variance):

- regularization
- get more training data (can't always get more training data / could be expensive to get more data

Adding regularization often help to prevent overfitting / reduce the errors in the NN

Adding L2 / L1 penalty term:

Cost function must be minimized!

$$cost\_r = cost + \text{penalty term}$$

L2 regularization

$$cost\_r = cost + \frac{\lambda}{2m}\sum\|w\|^2$$

L1 regularization

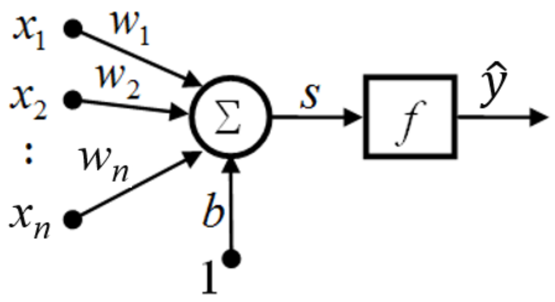
$$cost\_r = cost + \frac{\lambda}{2m}\sum\|w\|$$

λ – the regularization parameter

L2 regularization is also known as *weight decay* as it forces the weights to decay towards zero (but never zero).

For L1 regularization the weights may be reduced to zero.

Logistic regression
L2 & L1 regularization



$$s = w^T x + b \quad \hat{y} = f(s)$$

$$\hat{y} = f(w^T x + b)$$

Cost function across m examples is :

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

minimise $J(w,b)$ $_{w,b}$

- Add regularization:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

λ - regularization parameter

L$_2$ regularization – 2nd order

$$\|w\|_2^2 = \sum_{j=1}^{n} w_j^2 = w^T w$$ Euclidean norm

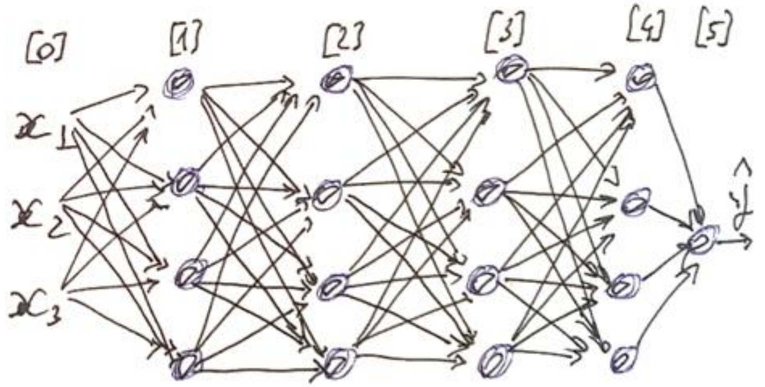
λ is a hyperparameter to be tuned.

L$_1$ regularization – 1st order

L$_1$ regularization: $+\frac{\lambda}{2m}\|w\|_1$

$$\|w\|_1 = \sum_{j=1}^{n} |w_j|$$

Neural network
L2 regularization



$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\left(\mathcal{L}(\hat{y}^{(i)}, \hat{y}^{(i)}\right)$$

$$\min_{w^{[l]}, b^{[l]}} J\left(w^{[l]}, b^{[l]}\right) \quad l=1,\dots L$$

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\left(\mathcal{L}(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2$$

Cost cross-entropy cost regularization cost

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{(l)}}\sum_{j=1}^{n^{(l-1)}}\left(w_{ij}^{[l]}\right)^2$$

$$w^{(l)} : (n^{(l)}; n^{(l-1)})$$

Frobenius norm (sum of squares of elements of a matrix)

Neural network - L_2 regularization   GDA implementation

- **Without regularization**

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$w^{[l]} := w^{[l]} - \eta\, dw^{[l]} \qquad dw^{[l]} = \frac{\partial J}{\partial w^{[l]}}$$

- **With regularization**

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \left(\mathcal{L}(\hat{y}^{(i)}, y^{(i)})\right) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

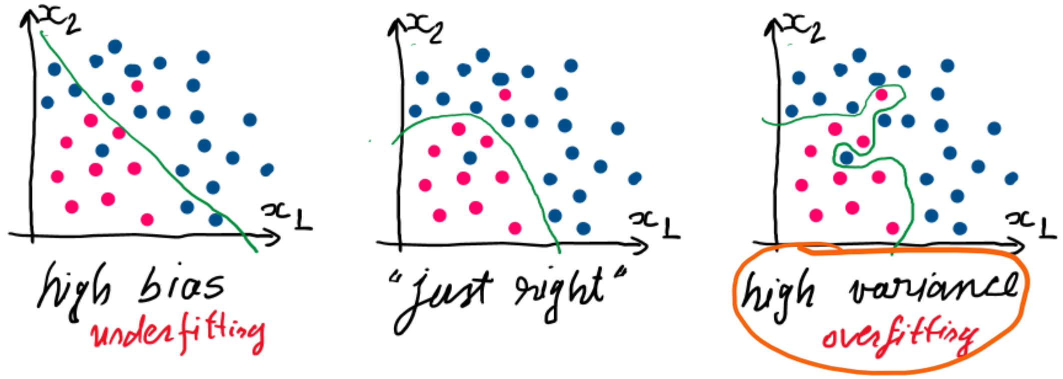
$$w^{[l]} := w^{[l]} - \eta\left(dw^{[l]} + \frac{\lambda}{m} w^{[l]}\right) = w^{[l]} - \frac{\eta\lambda}{m} w^{[l]} - \eta\, dw^{[l]}$$

$$w^{[l]} := \left(1 - \frac{\eta\lambda}{m}\right) w^{[l]} - \eta\, dw^{[l]}$$

Weight decay

- the coefficient in front of $w^{[l]} < 1$

Why L2 regularization reduces overfitting? (intuition)

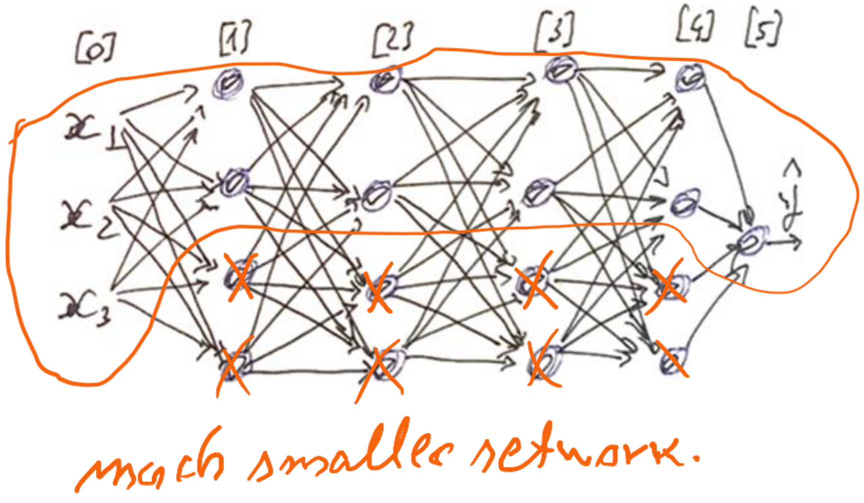


$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \left(\mathcal{L}(\hat{y}^{(i)}, y^{(i)})\right) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

Why does this term reduce overfitting?

The *w* weights are stimulated to become very small (close to zero), to minimize *J*

$w_{ij} \sim 0$ for a lot of hidden neurons



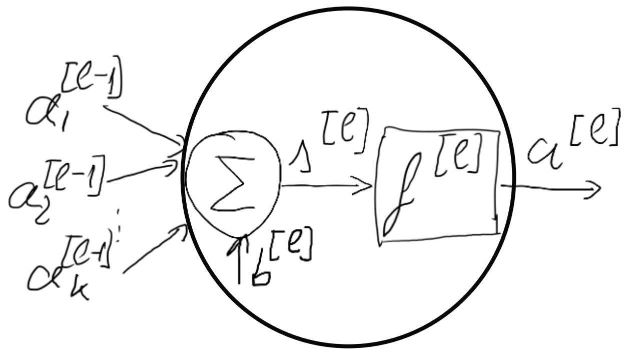
This highly simplified neural network (much smaller neural network) will take us from the overfitting case closer to the underfitting case (for large λ).

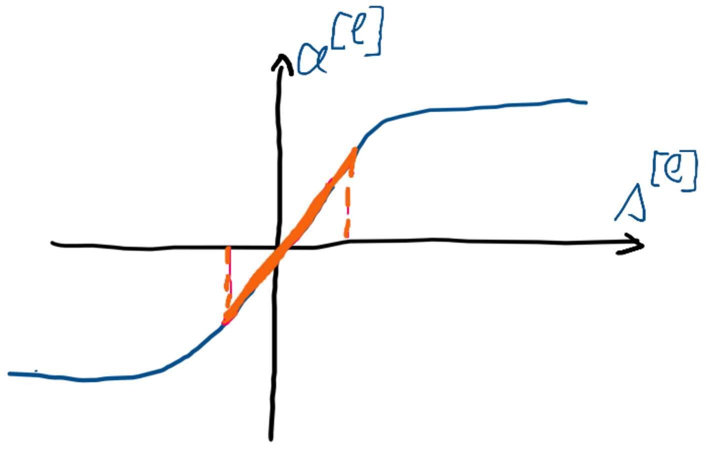
Hopefully, **there will be an intermediate value of λ that leads toward just right case.**

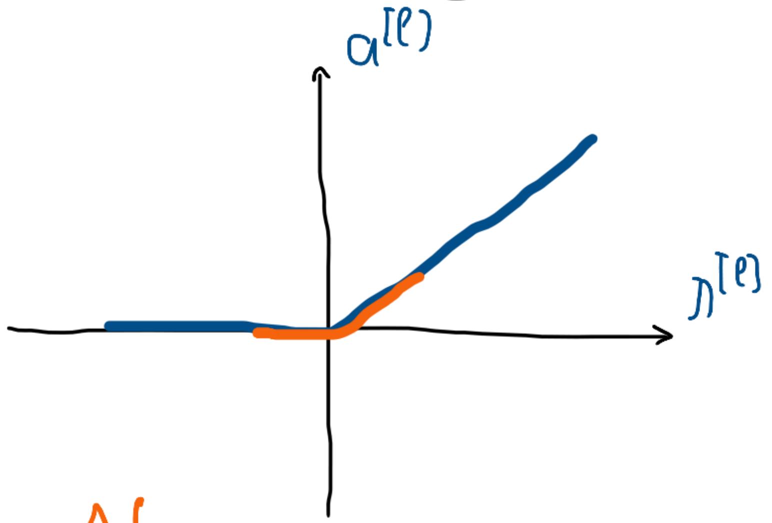
We can think of it as zeroing out or at least reducing the impact of a lot of the hidden units (especially the least significative weights).

Variance reduction

Why L2 regularization reduces overfitting? (intuition) – cont.







$\lambda \uparrow,\ w \downarrow,\ z \downarrow$

We are entering a narrow, almost linear region of the transfer function
This happens for all neurons, in all layers.

So, the **NN decreases its degree of nonlinearity**, approaching linearity and it cannot fit a verry complicated (highly non-linear) decision boundary

overfitting can hardly happen

Observations:

- The value of λ **is a hyperparameter that you can tune**.
- L2 regularization makes your decision boundary smoother.
  - If λ is too large, it is also possible to "over-smooth", resulting in a model with high bias.

L2-regularization relies on the assumption that **a model with small weights is simpler than a model with large weights**.

Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values.

It becomes too costly for the cost function to have large weights!

This leads to a smoother model in which the output changes more slowly as the input changes.

In *keras (python)*, we can directly **apply regularization to any layer**

Sample code to apply **L2 regularization to a Dense layer**.

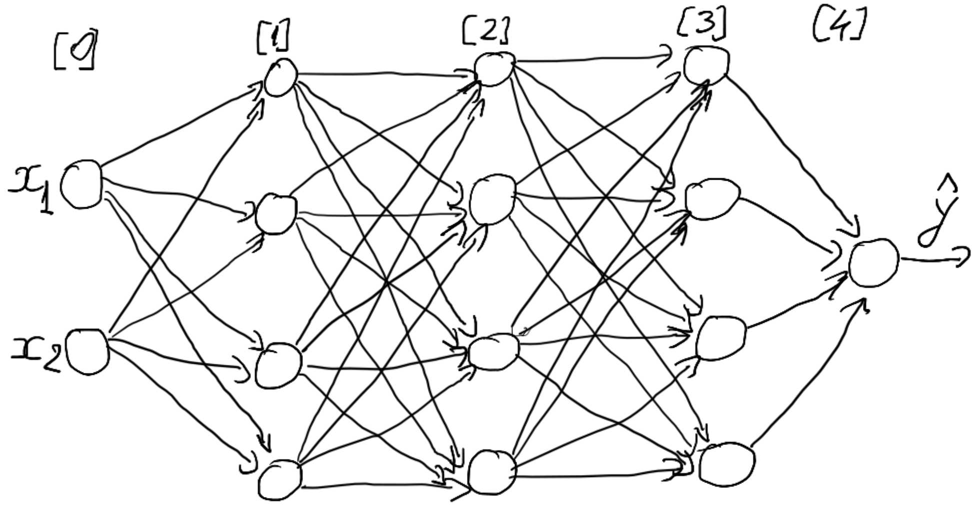
```
from keras import regularizers

model.add(keras.layers.Dense(64, activation = ‘relu’,
                    kernel_regularizer=keras.regularizers.l2(l2 = 0.01)
```

0.01 is the value of regularization parameter, i.e., lambda.

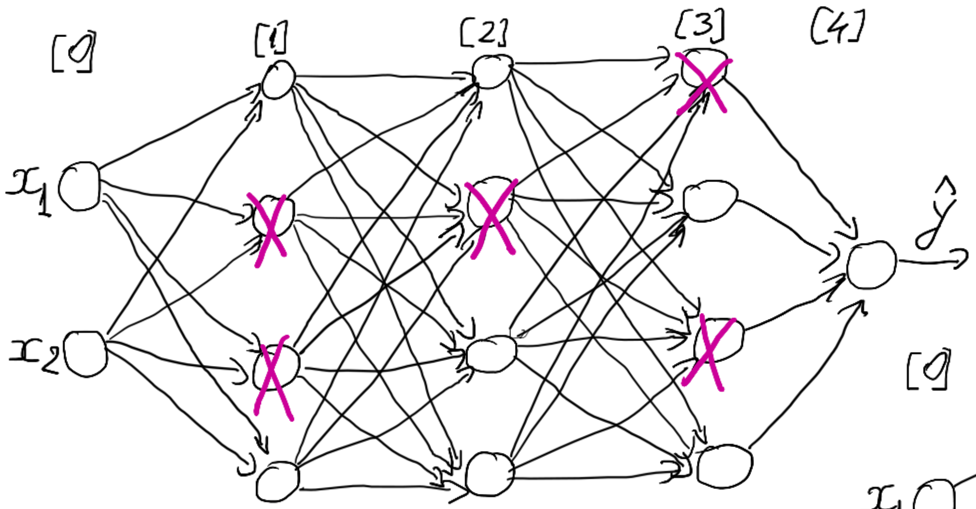
Sample code to apply **L1 regularization to a Dense layer**.

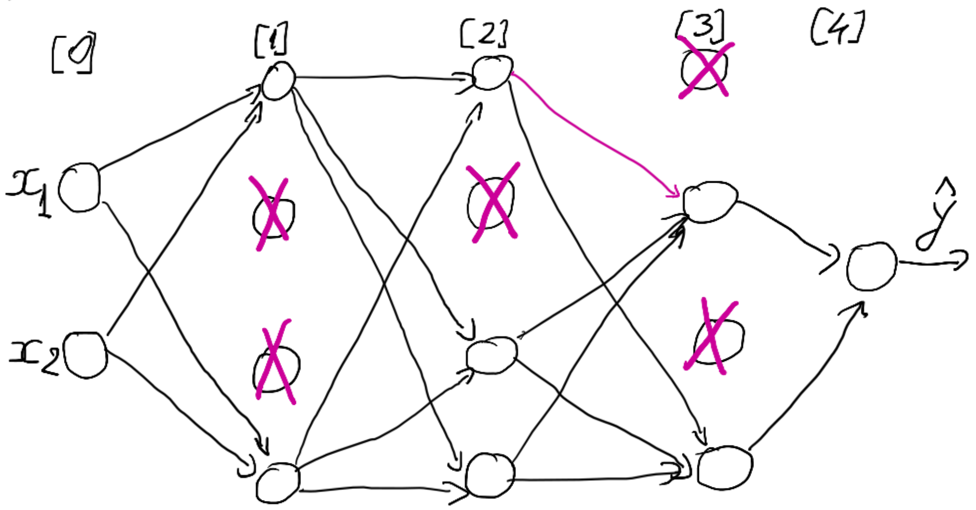
```
from keras import regularizers

model.add(keras.layers.Dense(64, activation = ‘relu’,
                    kernel_regularizer=regularizers.l1(l1 = 0.01)
```

Dropout regularization



At every iteration, dropout regularization **randomly** selects some nodes and removes them along with all their incoming and outgoing weights.





You end up with a much smaller, much diminished network.

Then you do back propagation training on this much diminished network.

Dropout regularization

Each training epoch has a different set of nodes, and this results in a different set of outputs.
It can also be thought of as an ensemble technique in machine learning.

Ensemble models usually perform better than a single model as they capture more randomness.
Similarly, dropout also performs better than a normal neural network model.

At each epoch, you shut down (= set to zero) each neuron of a layer with a certain probability (*keep_prob*)

The dropped neurons don't contribute to the training in both the forward and backward propagations of the current training epoch.

In each training epoch, only a part of the network weights are updated (those not connected to shut-down neurons), so that the possibility of overfitting (learning by heart the training data set) is considerably diminished.

The **probability** of choosing how many nodes should be dropped is the **hyperparameter of the dropout function**.

Dropout can be applied to both the hidden layers as well as the input layers.

Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

When you shut some neurons down, you actually modify your model.

The idea behind dropout is that at each iteration, you train a different model that uses only a subset of your neurons.

With dropout, your neurons thus become less sensitive to the activation of another specific neuron, because that other neuron might be shut down at any time.

Keras - Dropout layer

Dropout class

```
model.add(keras.layers.Dropout(0.2))
```

hyperparameter

The Dropout layer randomly **sets units to 0** with a frequency of **rate** at each step during training time, which helps prevent overfitting.

Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged (inverted dropout).

Note that the Dropout layer only applies when training is set to True such that no values are dropped during inference.

Dropout is inactive at inference time.

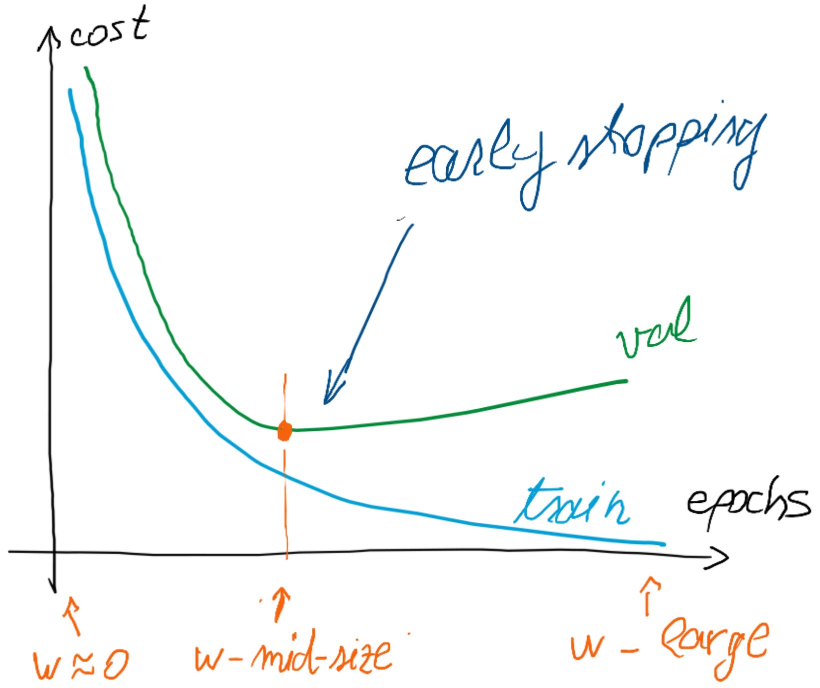
The trained network contains all neurons.

[https://keras.io/api/layers/regularization_layers/dropout/]

Other regularization techniques

Early stopping

The advantage of early stopping is that running the gradient descent process just once, you get to try out values of small *w*, mid-size *w*, and large *w*, without needing to try a lot of values of the L2 regularization hyperparameter lambda.



Rather than using early stopping, one alternative is just use L2 regularization, then you can just train the neural network as long as possible.

The downside of this: you might have to try a lot of values of the regularization parameter lambda. This makes searching over many values of lambda more computationally expensive.

Other regularization techniques Early stopping

A regularization technique used in deep learning to prevent overfitting. **Overfitting occurs when a model learns the training data too well, including its noise and random fluctuations, resulting in poor performance on unseen data**.

How it Works:

1. Validation Set: A portion of the training data is set aside as a validation set. This set is not used to train the model directly.

2. Monitoring: During training, the model's performance (e.g., loss or accuracy) is evaluated on both the training set and the validation set.

3. Stopping Criteria: If the model's performance on the validation set starts to worsen (e.g., validation loss increases or validation accuracy decreases) while the performance on the training set continues to improve, it indicates that the model is starting to overfit.

4. Early Stop: Training is stopped before the model has a chance to fully overfit the training data. The model parameters from the epoch with the best performance on the validation set are saved and used as the final model.

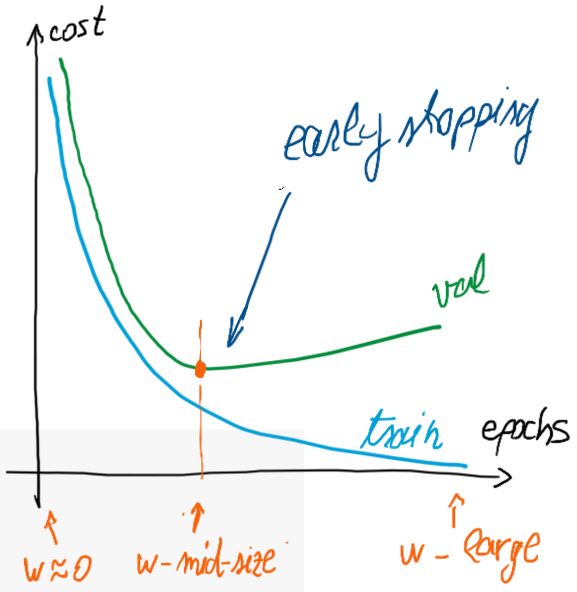
Benefits of Early Stopping:

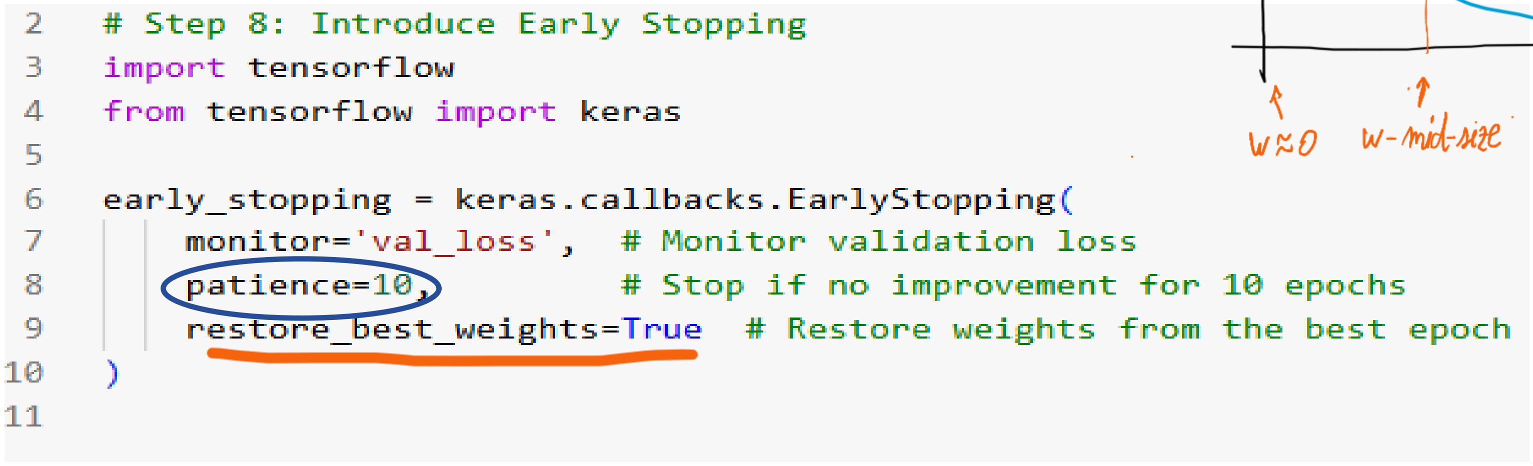
•**Prevents Overfitting:** Stops training before the model overfits, leading to better generalization on unseen data.

•**Saves Time and Resources:** Reduces unnecessary training time and computational resources by stopping training when further improvements are unlikely.

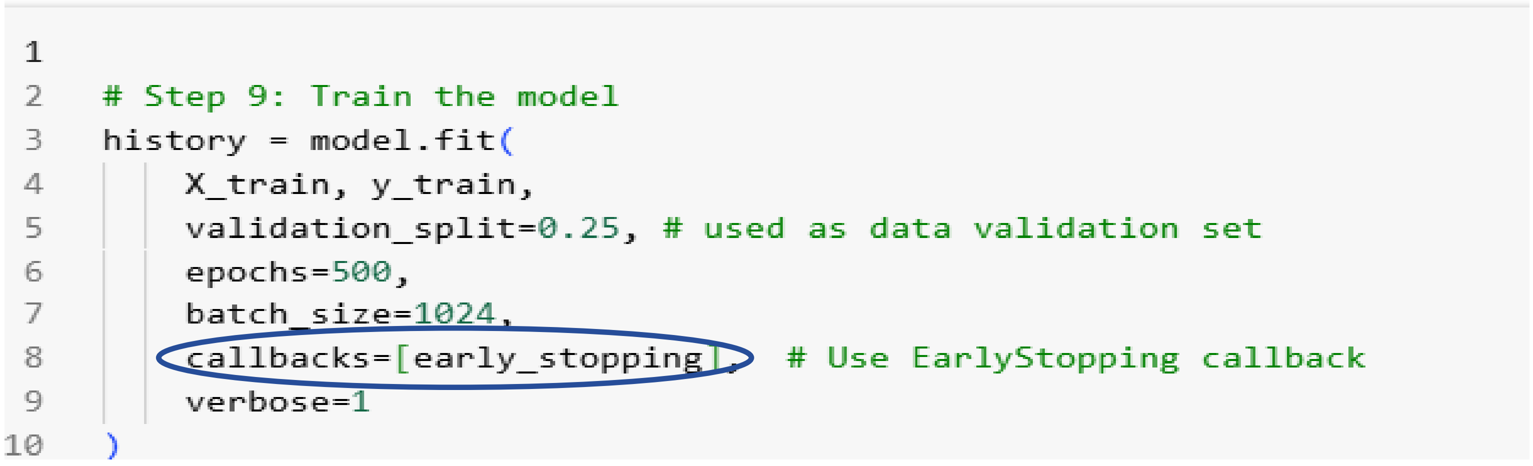
•**Improves Model Performance:** Can lead to a more robust and accurate model by selecting the best performing model during training.

ANN model with Early stopping regularization





```python
# Step 8: Introduce Early Stopping
import tensorflow
from tensorflow import keras

early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',  # Monitor validation loss
    patience=10,         # Stop if no improvement for 10 epochs
    restore_best_weights=True  # Restore weights from the best epoch
)

```



```python

# Step 9: Train the model
history = model.fit(
    X_train, y_train,
    validation_split=0.25, # used as data validation set
    epochs=500,
    batch_size=1024,
    callbacks=[early_stopping],  # Use EarlyStopping callback
    verbose=1
)
```

Setting up the optimization problem

- Data normalization
- Network (weights) initialization

Normalize dataset

Data **normalization** is often **a crucial step** in machine learning, especially for algorithms like the ANN

1. Feature Scaling and Algorithm Performance

- **Gradient Descent-Based Algorithms (like ANNs):** These algorithms use gradient descent to find the optimal model parameters.
  - If features have vastly different scales, the loss function will have an elongated shape, making it difficult for gradient descent to converge efficiently.
  - Normalization helps to create a more spherical loss function, leading to faster and more stable convergence.
- **Distance-Based Algorithms (like k-Nearest Neighbors):** These algorithms rely on calculating distances between data points.
  - Features with larger ranges can disproportionately influence distance calculations, potentially leading to inaccurate results.
  - Normalization ensures that all features contribute equally to distance measures.

Normalize dataset

2. Preventing Feature Dominance

- ➢ Without normalization, features with larger values can dominate the model's learning process, even if they are not necessarily more important. Normalization levels the playing field, allowing the model to learn relationships between all features more effectively.

3. Improving Numerical Stability

- ➢ In some cases, features with very large or very small values can lead to numerical instability during calculations. Normalization can help to mitigate these issues.

4. Enhanced Model Interpretability

- ➢ When features are normalized, the model's coefficients (or weights) become more comparable, making it easier to interpret the relative importance of different features.

Normalize dataset

Specific to ANN Model:

✓Normalizing the dataset features ensures that features with different scales do not disproportionately influence the weight updates during training. This leads to faster convergence and potentially better model performance.

In summary, normalization is often necessary to:

- ➢Improve the performance and stability of many machine learning algorithms.
- ➢Prevent feature dominance and ensure fair contribution from all features.
- ➢Enhance numerical stability during calculations.
- ➢Improve the interpretability of the model.

MinMaxScaler

- MinMaxScaler - preprocessing technique in scikit-learn used for feature scaling.
- It transforms features by scaling them to a given range, typically between 0 and 1.
- This process is also known as min-max normalization.

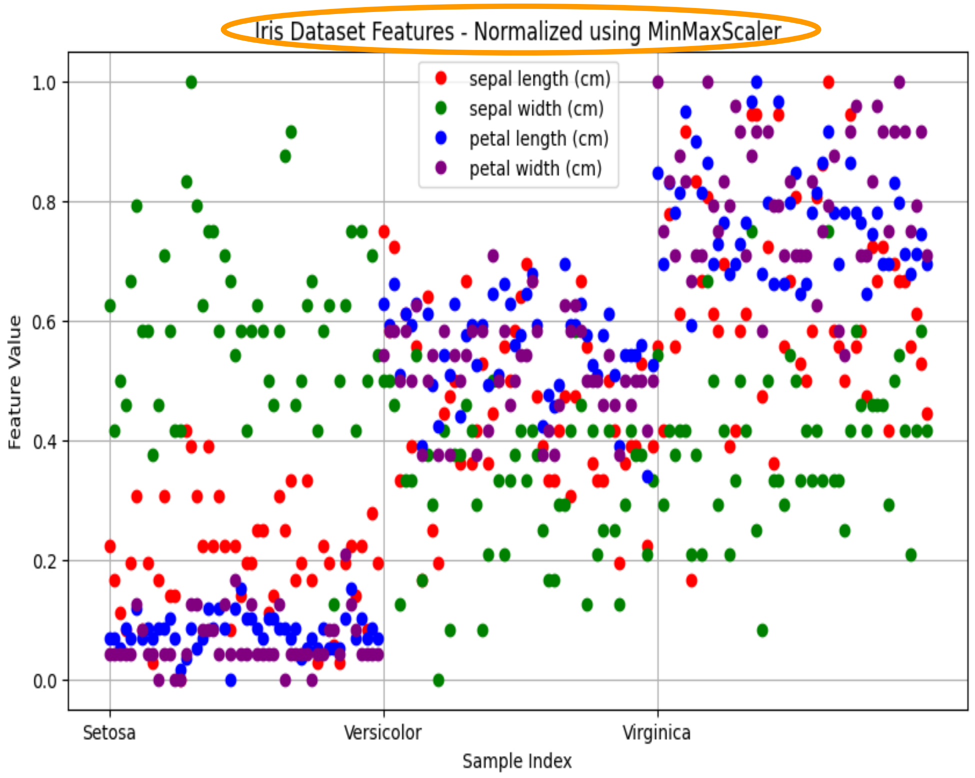
x_scaled = (x - x_min) / (x_max - x_min)

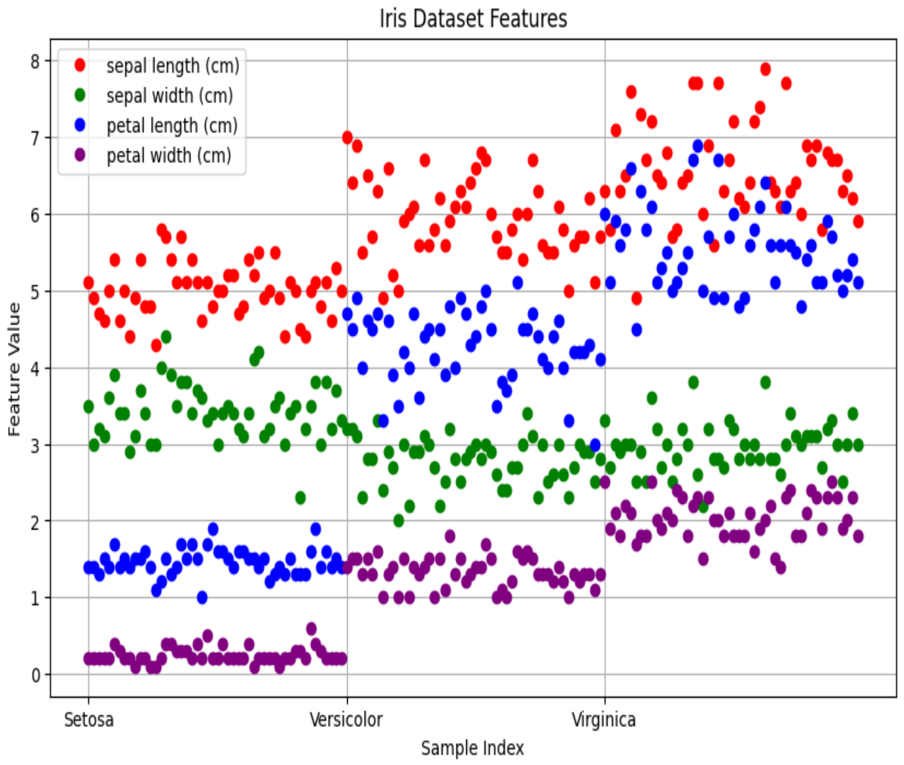
- ✓ **Preserves Data Shape**: MinMaxScaler preserves the original distribution of the data, meaning that the relative relationships between data points are maintained after scaling.
- ✓ **Handles Outliers**: Outliers can influence the scaling process, but their impact is limited since MinMaxScaler uses the minimum and maximum values of the entire dataset for scaling.
- ✓ **Simple and Intuitive**: The scaling process is easy to understand and interpret.
- ✓ **Suitable for Algorithms Sensitive to Feature Ranges**: Algorithms like k-Nearest Neighbors, Support Vector Machines, and Neural Networks often benefit from feature scaling using MinMaxScaler.

- ▪ **Sensitive to New Data:** If new data points with values outside the original range are introduced, the scaler needs to be refitted to include these values, which can affect the scaling of existing data.
- ▪ **May Squash Data:** If the data has a wide range, MinMaxScaler can compress the data into a smaller range, potentially losing some information.

MinMaxScaler

```python
from sklearn.preprocessing import MinMaxScaler  # Import MinMaxScaler
# Normalize the data using MinMaxScaler
scaler = MinMaxScaler()
X_normalized = scaler.fit_transform(X)
```

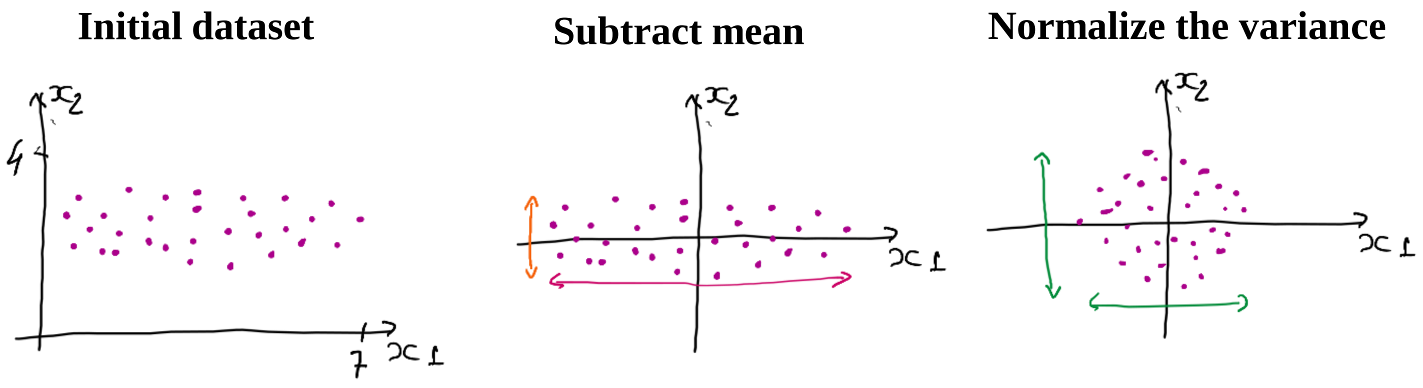




Standard Scaler

$$x\_scaled = \frac{x - x\_mean}{standard\_deviation}$$

Applied separately on each data feature



Use the same μ, σ to normalize all data sets

- ✓Training
- ✓Validation
- ✓Test

$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$; $\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$

$x := x - \mu$

μ – mean

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x^{(i)})^2$

σ^2 – variance

$x := \frac{x}{\sigma}$

σ – standard deviation $\sigma = \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$

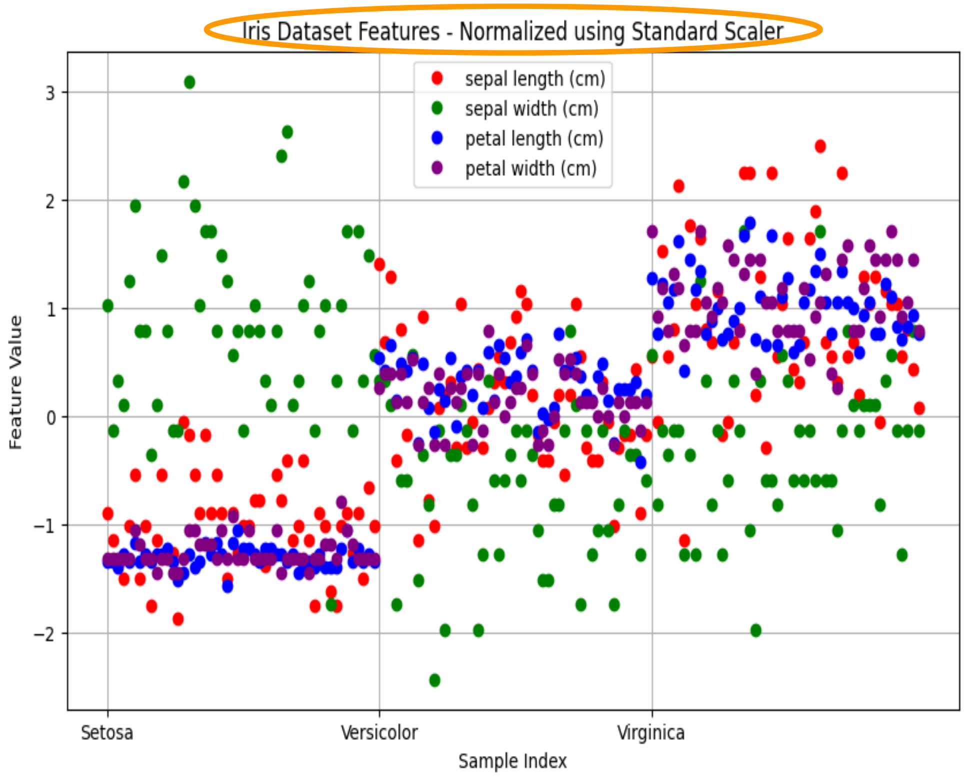
Standardizes features by removing the mean and scaling to unit variance.

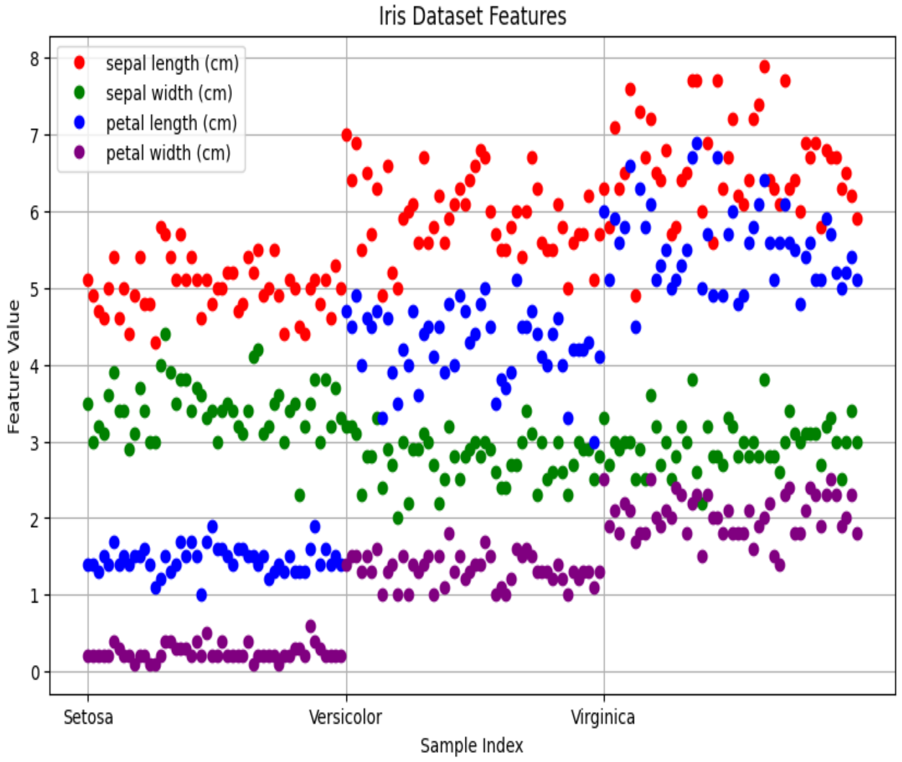
1. **Centering:** The mean of the feature is subtracted from each feature value (x). This shifts the distribution of the feature so that its mean becomes 0.
2. **Scaling:** Each centered feature value is then divided by the standard deviation. This scales the distribution so that its variance becomes 1.

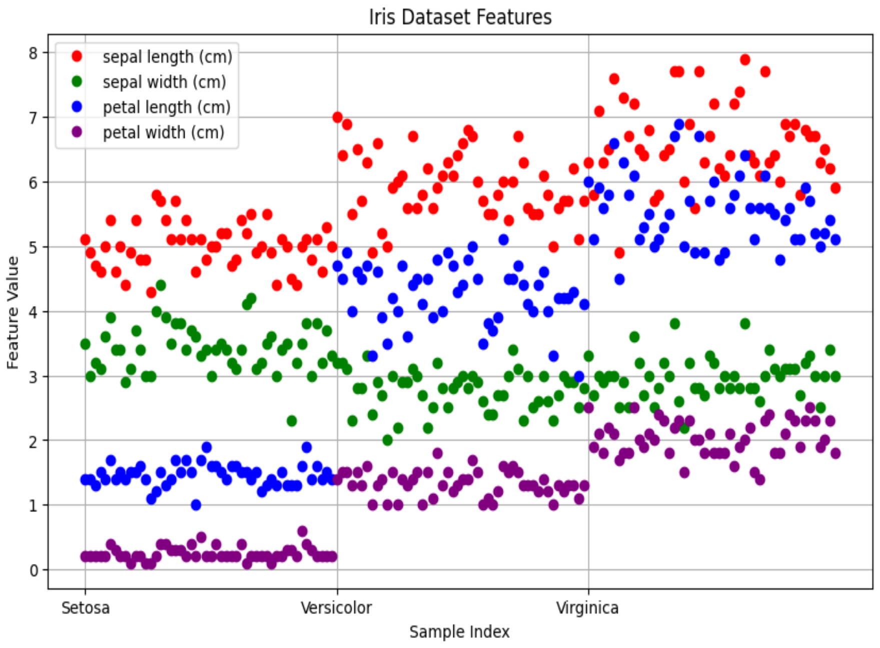
```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_normalized_ss = scaler.fit_transform(X)
```

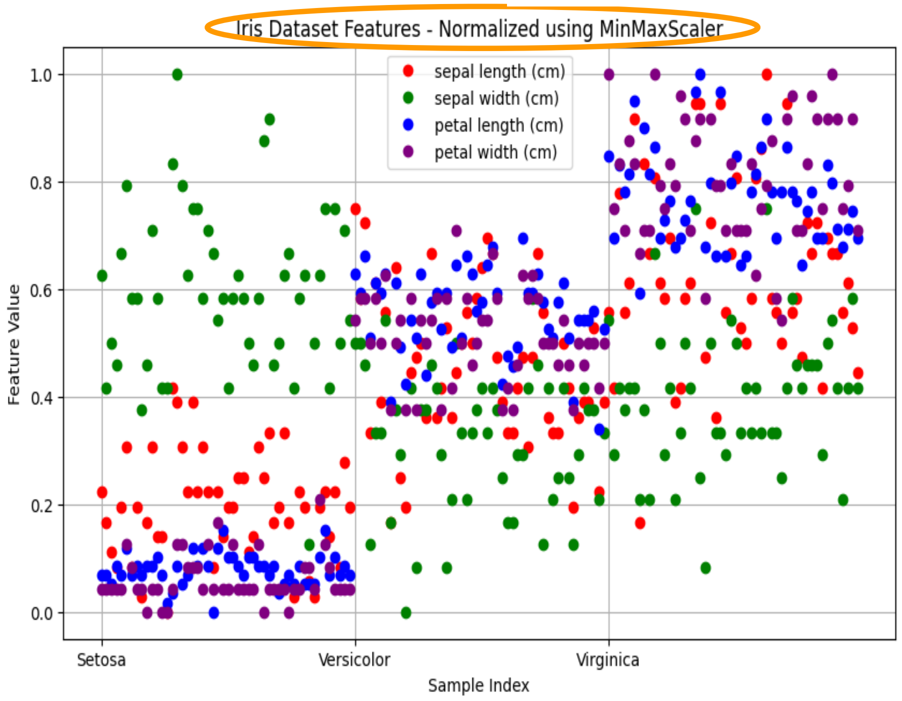
Standard Scaler

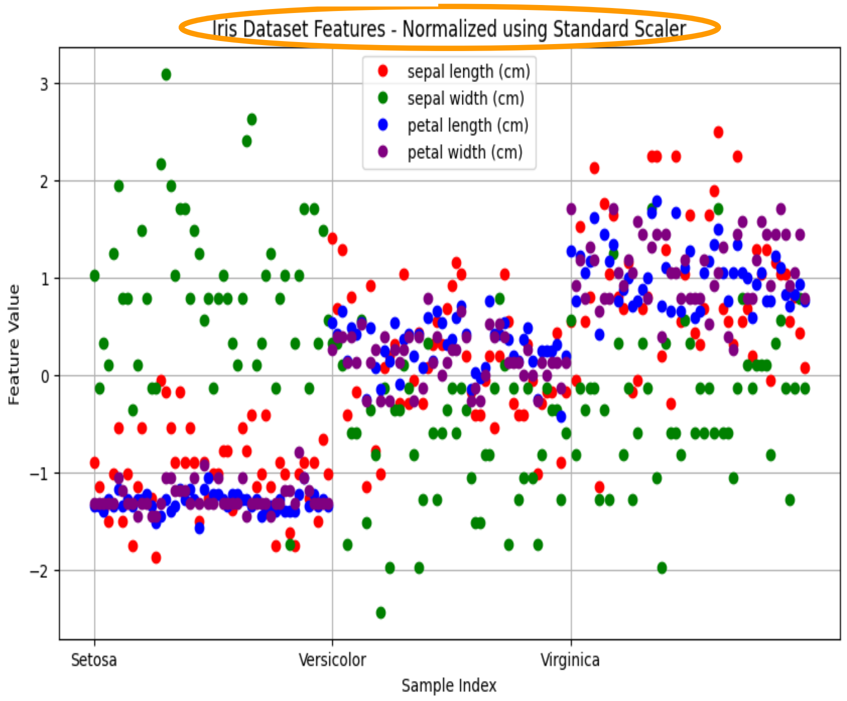
```
sepal length (cm): Mean: 5.84 Std_dev: 0.83
sepal width (cm):  Mean: 3.06 Std_dev: 0.43
petal length (cm): Mean: 3.76 Std_dev: 1.76
petal width (cm):  Mean: 1.20 Std_dev: 0.76
```





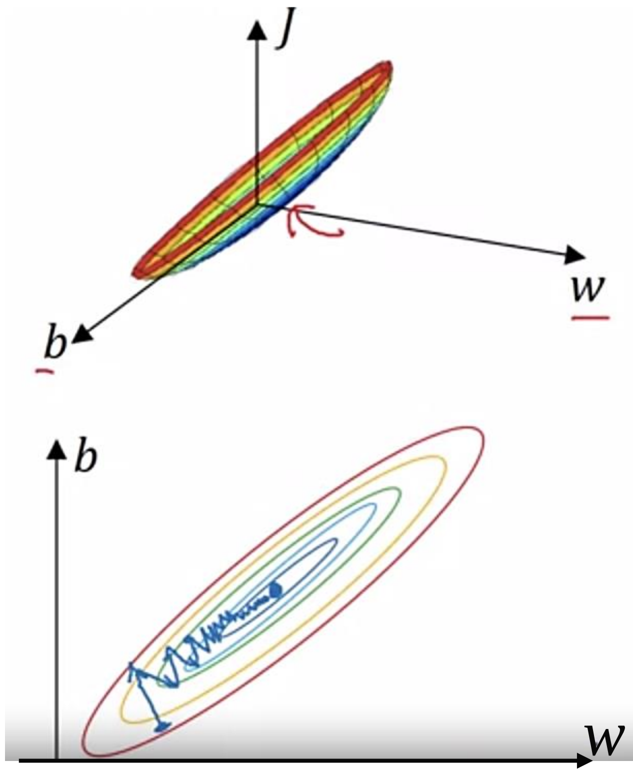

Iris Dataset Features
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
Feature Value
Setosa
Versicolor
Virginica
Sample Index

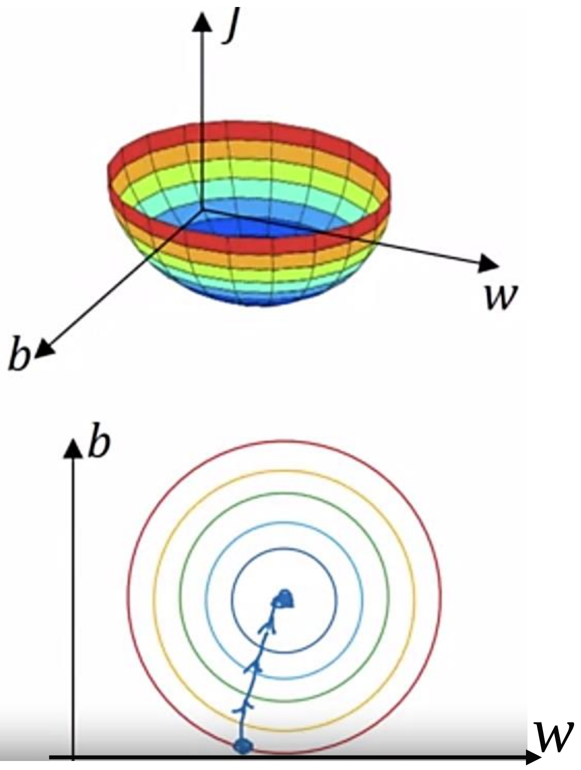

Iris Dataset Features - Normalized using MinMaxScaler
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
Feature Value
Setosa
Versicolor
Virginica
Sample Index


Iris Dataset Features - Normalized using Standard Scaler
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
Feature Value
Setosa
Versicolor
Virginica
Sample Index

Why normalization?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$





Unnormalized

Very low learning rate, a lot of steps

Normalized

Go straight to the minima

J is easier and faster to optimize

[DeepLearning.AI, Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization, https://www.coursera.org/learn/deep-neural-network/home/welcome]

Vanishing / exploding gradients – network initialization

(Very) Deep neural network have a **major setback**

- **vanishing gradient**
- **exploding gradient**

Exploding gradient

In deep networks, **error gradients can accumulate** during an update and result in very large gradients. The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.

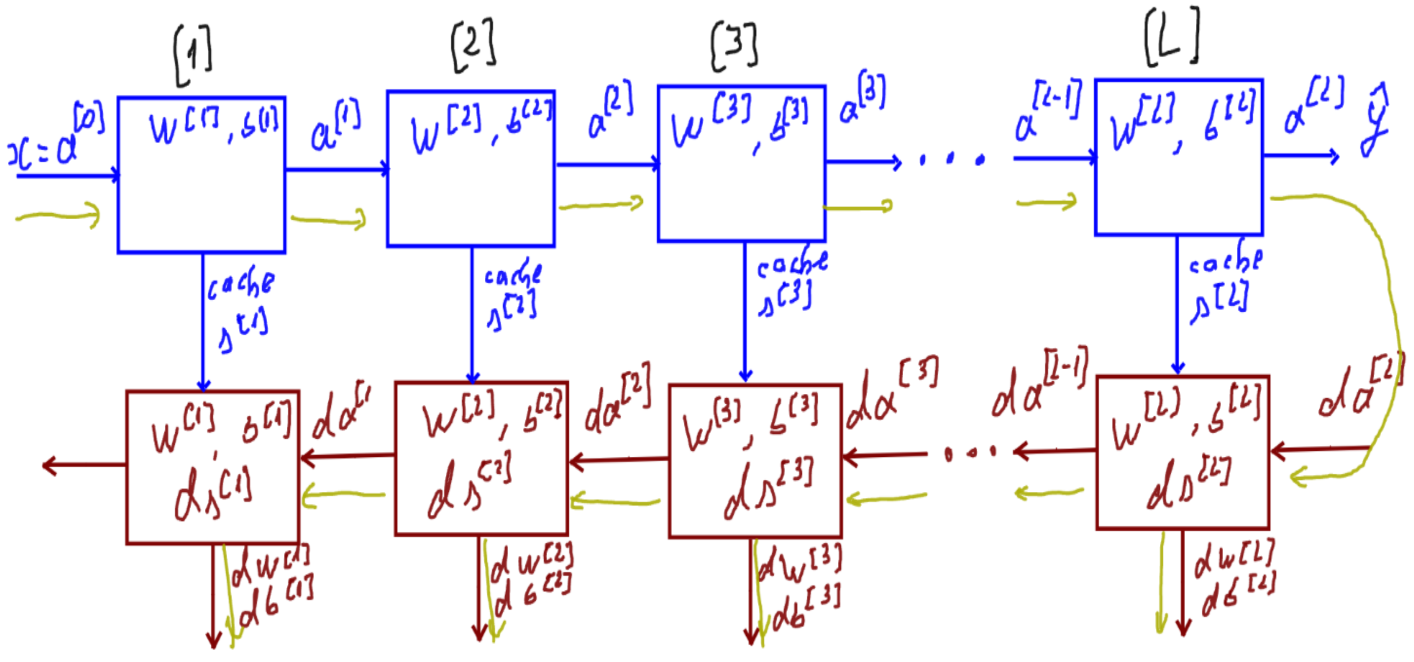
These in turn result in large updates to the network weights, and in turn, an unstable network. At an extreme, the values of weights can become so large as to overflow and result in NaN values.

Vanishing gradient

When *n* hidden layers use an activation that give small gradients (below unity, like the sigmoid function), *n* small derivatives are multiplied together. Thus, the **error gradient decreases exponentially** as we propagate down to the initial layers.

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.

Forward and backward propagation for a DNN



— forward prop
— backward prop
— full iteration

$$w^{[l]} := w^{[l]} - \eta\, dw^{[l]}$$
$$b^{[l]} := b^{[l]} - \eta\, db^{[l]}$$

η - learning rate

For each layer $w := w - \eta\, dw$ $\quad dw = \dfrac{\partial \mathcal{L}}{\partial w} = \dfrac{\partial \mathcal{L}}{\partial z} \cdot \dfrac{\partial z}{\partial w} = \dfrac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \dfrac{\partial \hat{y}}{\partial z} \cdot \dfrac{\partial z}{w}$

For multiple layer – multiplications accumulate for all layers

Vanishing / exploding gradients – network initialization

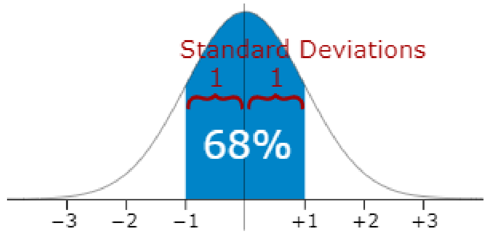
Partial solution – careful choice of the **random initialization of the network** (initial weights)

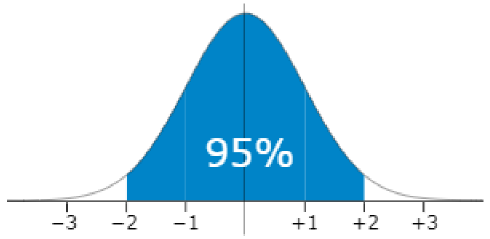
$$W^{[l]} = \text{np.random.randn}(n^{[l]}, n^{[l-1]}) * \text{np.sqrt}\left(\frac{cst}{n^{[l-1]}}\right)$$

standard normal distribution
(mean = 0, standard deviation = 1)

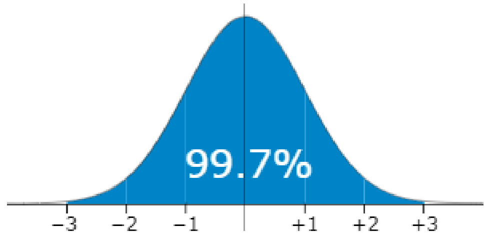
Introduces a variance that depends on the number of input features for the layer

Can be seen as a hyperparameter to be tuned





$cst = 2$ for *ReLU* activation function
$cst = 1$ for *tanh* activation function



Hopefully, that makes the weights not explode too quickly and not decay to zero too quickly, so you can train a reasonably deep network without the weights or the gradients exploding or vanishing too much.

Overfitting and regularization - recommended reading

http://neuralnetworksanddeeplearning.com/chap3.html#overfitting_and_regularization

Setting a DNN –recommended exercise

http://playground.tensorflow.org/#activation=relu&regularization=L2&batchSize=5&dataset=xor&regDataset=reg-gauss&learningRate=0.03&regularizationRate=0.001&noise=20&networkShape=2&seed=0.93433&showTestData=false&discretize=true&percTrainData=70&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

Augmentation, L2 regularization, dropout implementation – recommended programimg exercise

https://colab.research.google.com/drive/1moK2cq2SSgJLB68uNyvjyrGQKjwh8hKU?usp=sharing

To download the dataset:

https://drive.google.com/drive/folders/10HMkJbgl0XVtxGngP7NS1uWM9LbgrGez?usp=sharing

Tuning process

The processes that drive performance and generate good results systematically

- **Hyperparameters:**
  - Learning rate – most important
  - Learning rate decay
  - Mini-batch size
  - Momentum term; hyperparameters of the optimization algorithms
  - Number of layers
  - Number of hidden units
- **Regularization method and params**