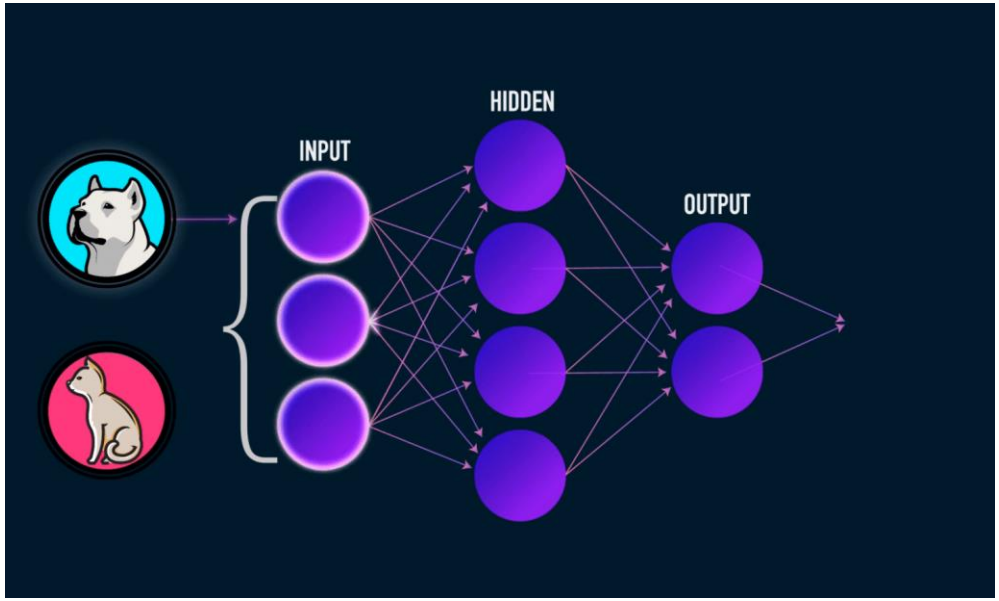
Neural Network with TensorFlow & Keras

OUTLINE

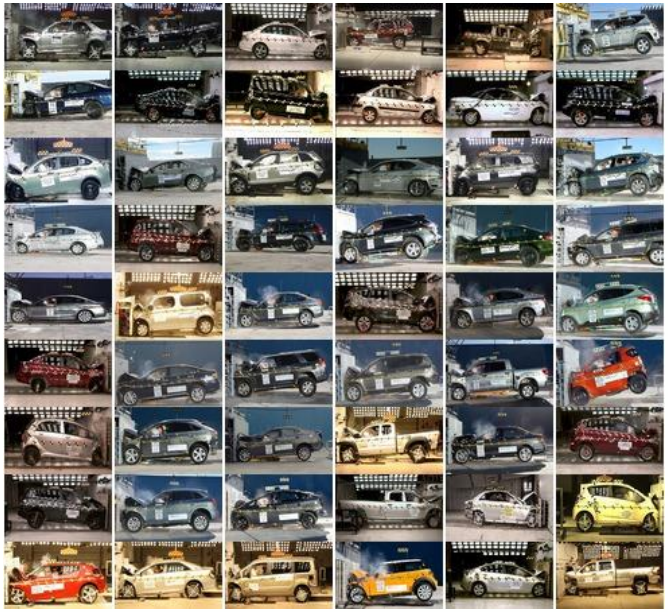
- Steps to train a Neural Network
- TensorFlow & Keras
- Evaluating DL Models
- Data Preparation
- Improving DL Models
- Alternative Framework: PyTorch

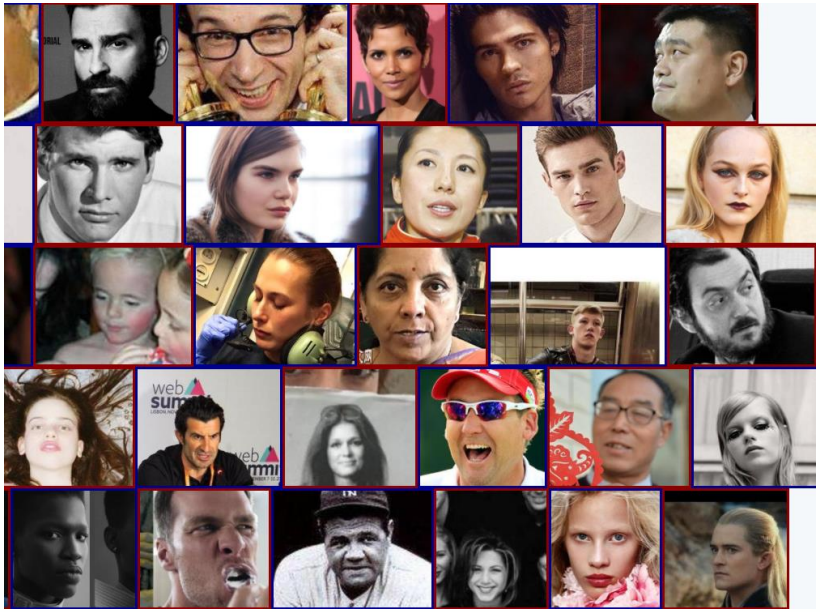
Steps to train a Neural Network

Steps to Train a Neural Network



Step 1: Collect your dataset



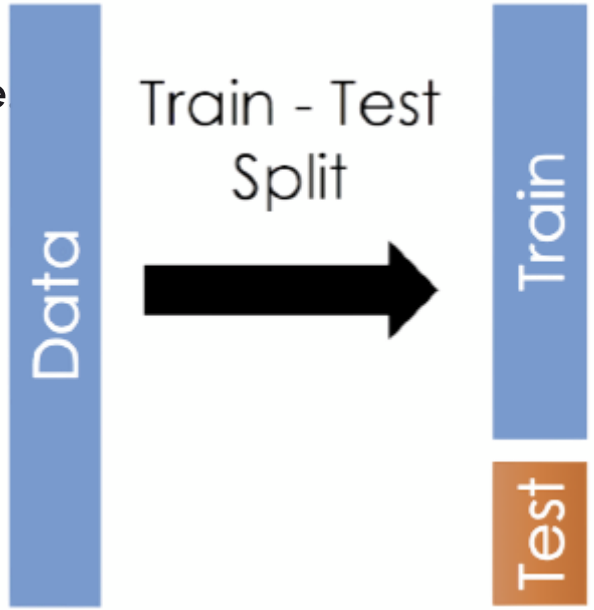


Step 2: Split the Data

We need the network to be able to generalize

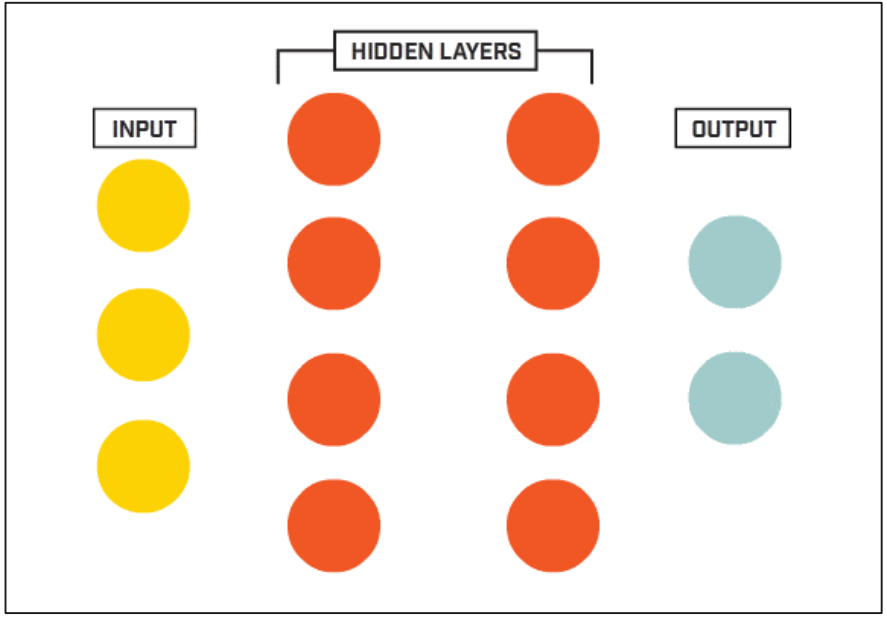
Split the training dataset:

- Train set
- Test set



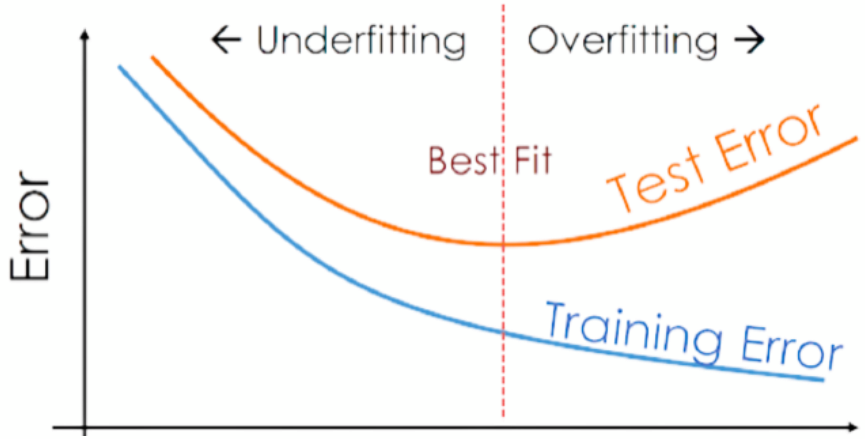
Step 3: Build and train the Neural Network

- Choice of ML framework/library
- Data pipeline & preparation
- Model architecture
- Network hyperparameters
  - Learning rate
  - Activation functions
  - Batch size
  - Epoch
  - Others
- TRAIN network!



Step 4: Evaluate the Model

- Underfitting
- Overfitting
- Good fitting



Step 5: Apply changes when needed

- Model architecture (network topology): add/remove layers, add/remove neurons, etc..
- Change activation functions (Sigmoid, tanH, RELU, etc...)
- Adjust Hyperparameters (learning rate, momentum, etc...)
- Adjust batch sizes and epochs
- Get more data!
- RESTART TRAINING and repeat from step 3

TensorFlow & Keras

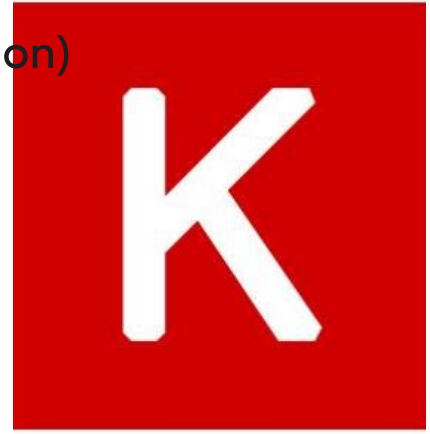
TensorFlow

- Python library for fast numerical computing
- Created and released by Google
- Open source

pip install tensorflow

Keras

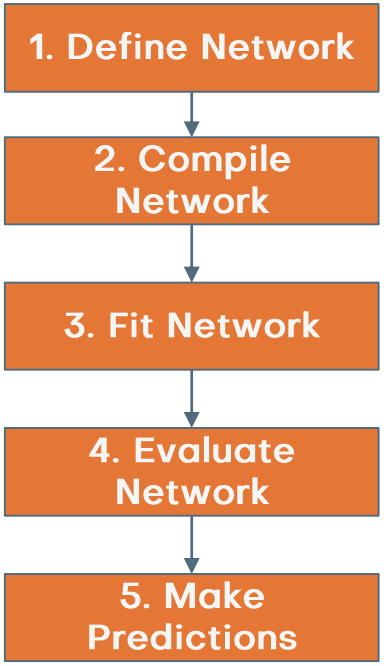
- Minimalist Python library for deep learning
- Can run on top of Theano or TensorFlow (old version)
- Merged with TensorFlow starting from TF 2.0
- Open source



Keras' Model Lifecycle

Below is an overview of the 5 steps in the neural network model life-cycle in Keras:

1. Define Network.
2. Compile Network.
3. Fit Network.
4. Evaluate Network.
5. Make Predictions.



Step 1: Define the Architecture

The first layer in the network must define the number of inputs to expect.

Different activation functions for the output layer:

- **Regression**: Linear activation function, or *linear* and the number of neurons matching the number of outputs.
- **Binary Classification** (2 classes): Logistic activation function, or *sigmoid*, and one neuron in the output layer.
- **Multiclass Classification** (>2 classes): Softmax activation function, or *softmax*, and one output neuron per class value, assuming a one hot encoded output pattern.

Step 2: Compile the Network

Compilation transforms the simple **sequence of layers** that we defined into a **highly efficient series of matrix transforms** in a format intended to be executed on your GPU or CPU.

Compilation requires a number of parameters to be specified:

- the **optimization algorithm** to use to train the network
- the **loss function** used to evaluate the network that is minimized by the optimization algorithm.

model.compile(optimizer="sgd", loss="mean_squared_error")

Step 2: Compile the Network

Different loss functions:

- **Regression**: Mean Squared Error or *mean_squared_error.*
- **Binary Classification**: Logarithmic Loss, also called cross-entropy or *binary_crossentropy*.
- **Multiclass Classification**: Multiclass Logarithmic Loss or *categorical_crossentropy*.

Step 3: Fit the Network

Fitting the network requires training data: inputs X and the outputs Y.

model.fit(X, Y, batch_size=10, epochs=100)

The network is trained using the backpropagation algorithm and optimized according to the **optimization algorithm** and **loss function** specified when compiling the model.

Step 4: Evaluate the Network

After the model is trained, we need to evaluate it to check its performance. This is done on a **SEPARATE DATASET!**

loss = model.evaluate(X_test, Y_test)

Step 5: Make Predictions

predictions = model.predict(X)

- **Regression**: predictions returned in the format of the problem directly
- **Binary classification**: prediction is a probability for the first class that can be converted to a 1 or 0 by rounding.
- **Multiclass classification**: an array of probabilities that need to be converted to a single class output prediction using the **argmax()** function

Keras: Sequential vs. Functional

The **sequential API** :

- create models layer-by-layer
- It is limited (can't create models that share layers or have multiple input or output layers)

The **functional API**:

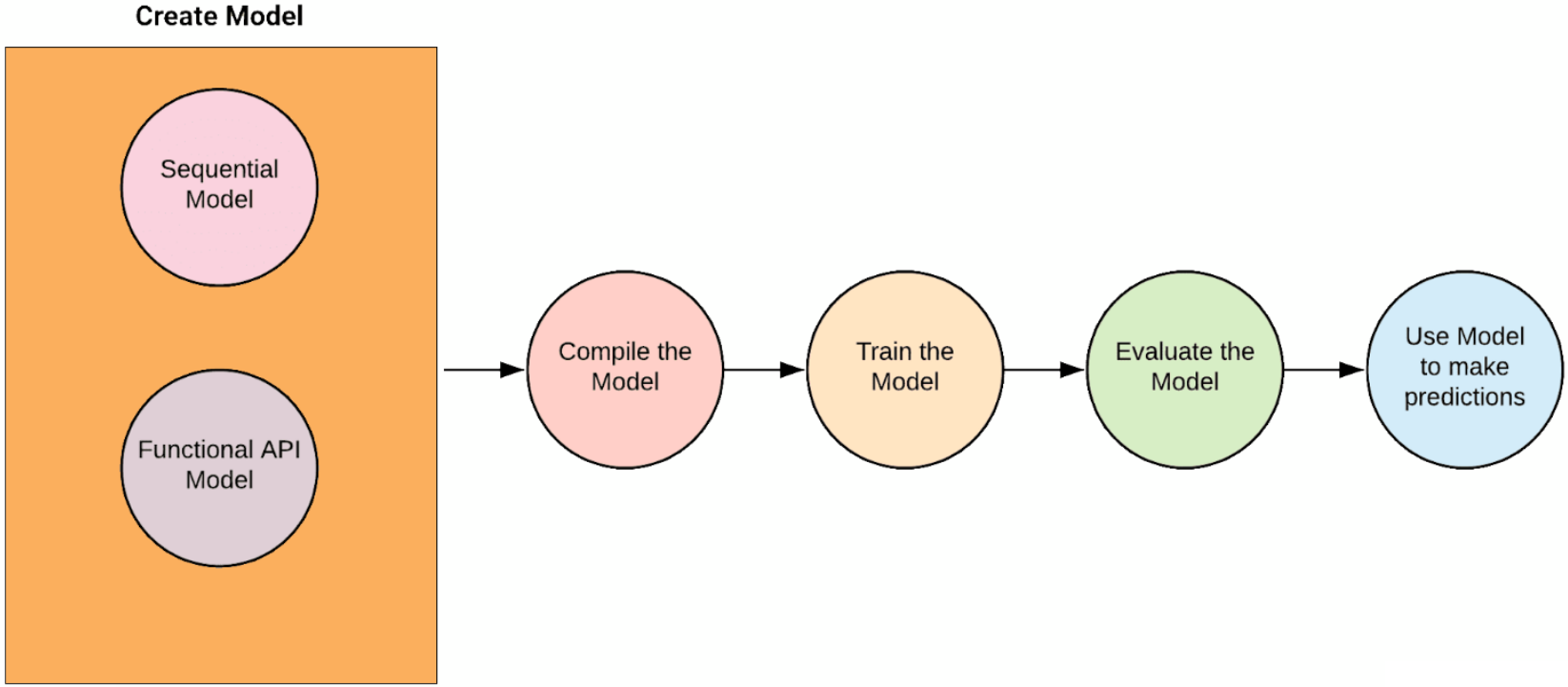
- An alternate way of creating models
- Offers a lot more flexibility, including creating more complex models.

Models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a Model that specifies the layers as the input and output.

Keras: Sequential vs. Functional

```
#sequential
model = Sequential()
model.add(Dense(10, input_dim = 3, activation = 'relu'))
model.add(Dense(10, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))

#functional
visible = Input(shape=(3,))
hidden1 = Dense(10, activation = 'relu')(visible)
hidden2 = Dense(10, activation = 'relu')(hidden1)
outlayer = Dense(1, activation = 'sigmoid')(hidden2)
model = Model(inputs = visible, outputs = outlayer)
```

Keras: Sequential vs. Functional



Hands-on: Sequential vs. Functional API with Keras

Evaluating Deep Learning Models

Data Splitting

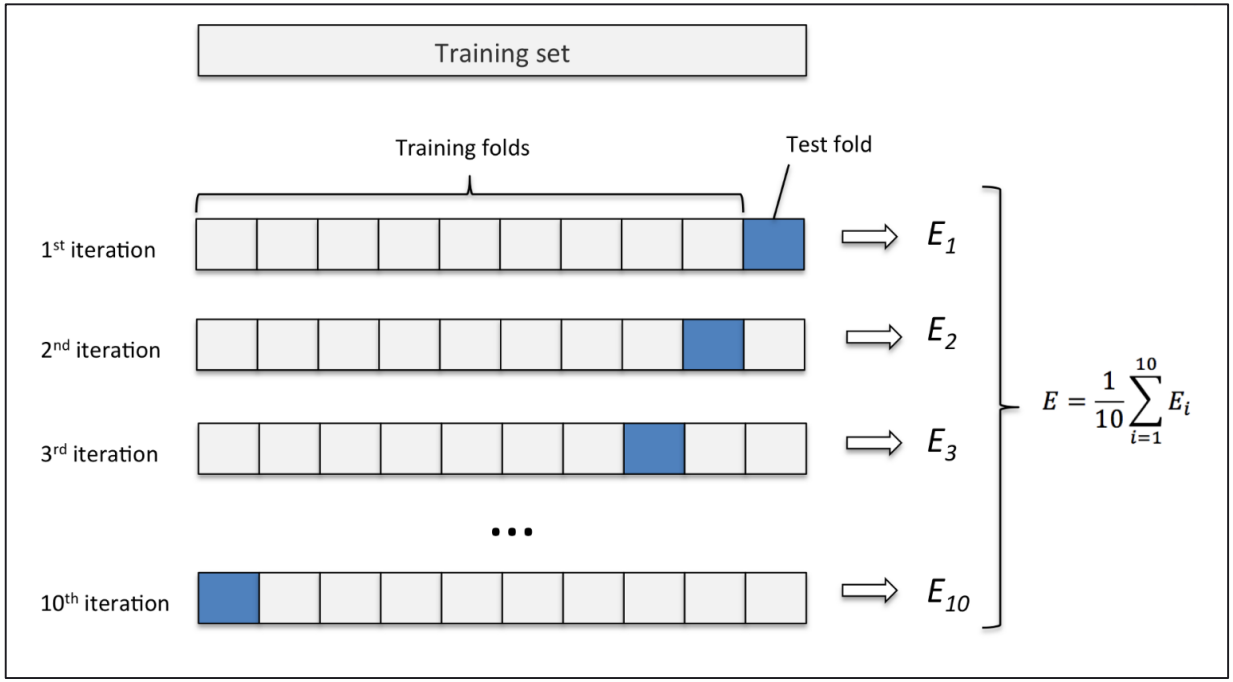
1. Use an automatic verification dataset.

```
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10)
```

2. Use a manual verification dataset.

```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33,
random_state=seed)

# Fit the model
model.fit(X_train, y_train, validation_data=(X_test,y_test),
epochs=150, batch_size=10)
```

Cross Validation



Scikit-learn & Keras

How to wrap Keras models so that they can be used with the scikit-learn machine learning library?

The **Keras** library provides a convenient wrapper for deep learning models to be used as classification or regression estimators in scikit-learn.

KerasClassifier and *KerasRegressor*

Define a function that creates a model and pass it to the Keras wrapper

Scikit-learn Cross Validation

1. Define function ***create_model()***

2. Create **KerasClassifier** model

   *model = KerasClassifier(build_fn=**create_model**, epochs=150, batch_size=10, verbose=0)*

3. Use **StratifiedKFold** class from the scikit-learn library

   # define 10-fold cross validation test harness
   *kfold = StratifiedKFold(n_splits=**10**, shuffle=True, random_state=seed)*

4. Evaluate using 10-fold cross validation

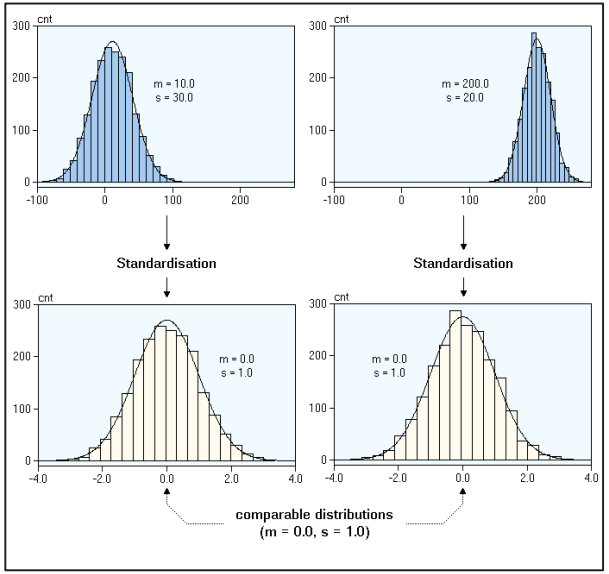
   results = cross_val_score(model, X, Y, cv=kfold)

Data Preparation

Data Preparation

Numerical values (age, height, price, etc..) => Rescale data!

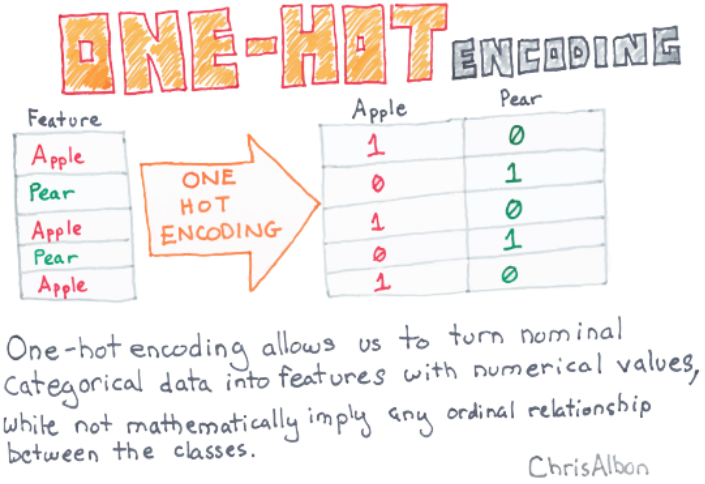
- MinMaxScaler
- StandardScaler



Data Preparation

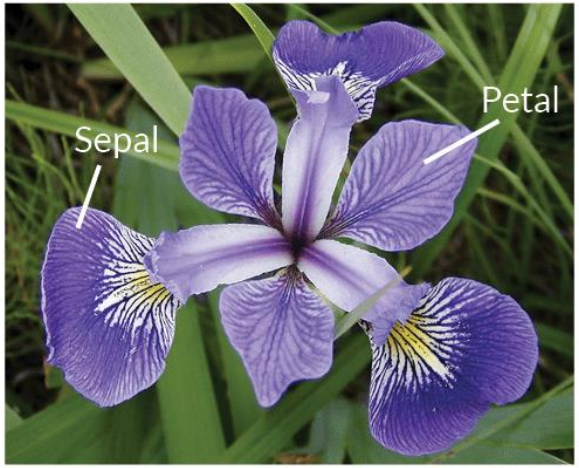
Categorical values (gender, color, animal, etc…)

- One-Hot encoding



Iris Dataset

Output values = 3 → Multiclass Classification!



Iris Versicolor

Iris Setosa

Iris Virginica

One-hot Encoding

Iris-Setosa	Iris-Versicolor	Iris-Virginica
1	0	0
0	1	0
0	0	1

```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```

Scikit-learn Pipeline

Use pipeline to automate standard applied machine learning workflows.

```
# evaluate baseline model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, epochs=100,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
```

Improving DL Models: Data Methods

Importance of Data

Data is the core of any DL model we want to build.

Data

Machine Learning

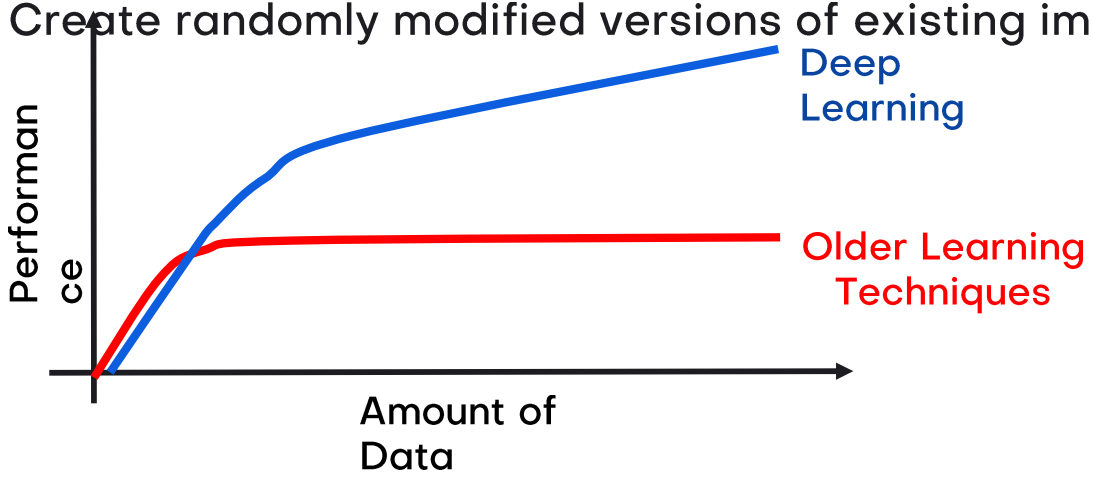
Build a Model

Predictive Outcome

Data Augmentation

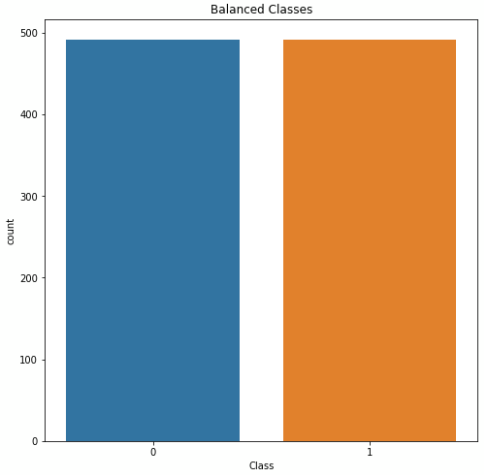
If you can't reasonably get more data, you can create more data.

- **Vectors of numbers?** Create randomly modified versions of existing vectors.
- **Images?** Create randomly modified versions of existing images.



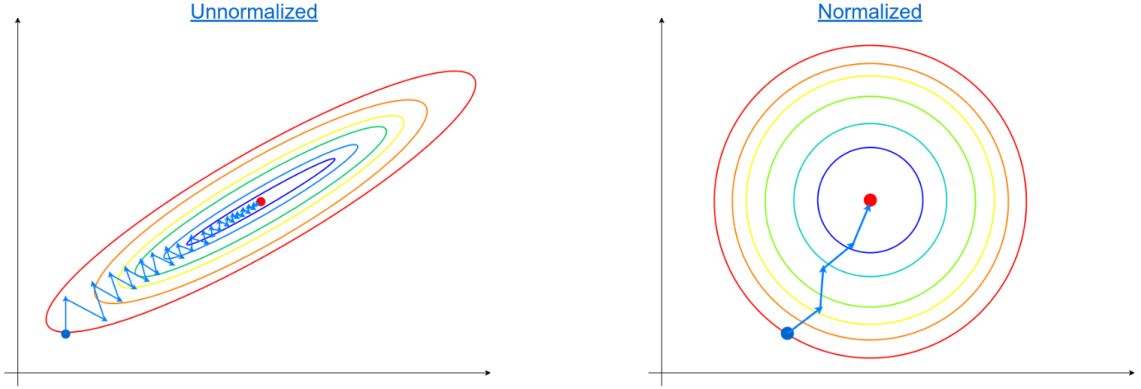
Balancing your Dataset

- **Undersample Majority Class:** You can balance the class distributions by subsampling the majority class.
- **Oversample Minority Class:** Sampling with replacement can be used to increase your minority class proportion.



Rescaling Data

- **Features** that are measured at different scales do not contribute equally to the analysis and might end up creating a bias.
- Having features on a similar scale can help the gradient descent converge more quickly towards the minimum.



Rescaling Data

To rescale the data, we have 2 techniques:

- Normalization
- Standardization

$$x_{normalized} = \frac{x}{x_{\max}}$$

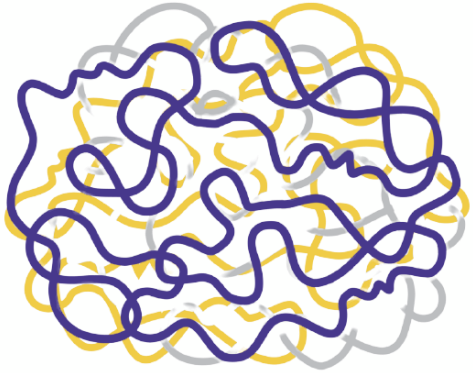
$$x_{standardized} = \frac{x - \mu}{\sigma}$$

The terminology of standardizing and normalizing is sometimes used interchangeably

Feature Selection

- **Feature Selection:** Selecting **features** which contribute most to your output.

Train faster



Reducing Complexity

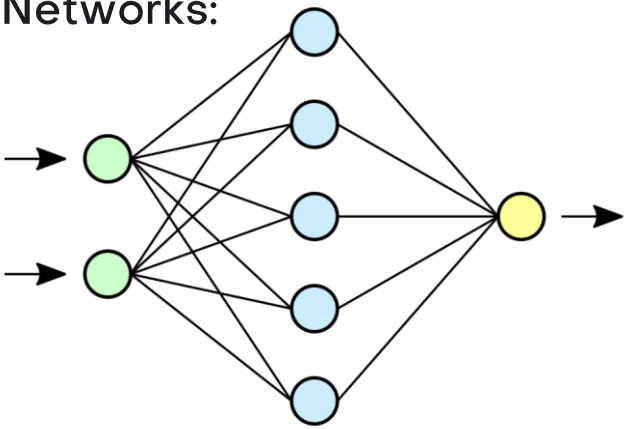
Improves accuracy

Improving DL Models: Model Design Methods

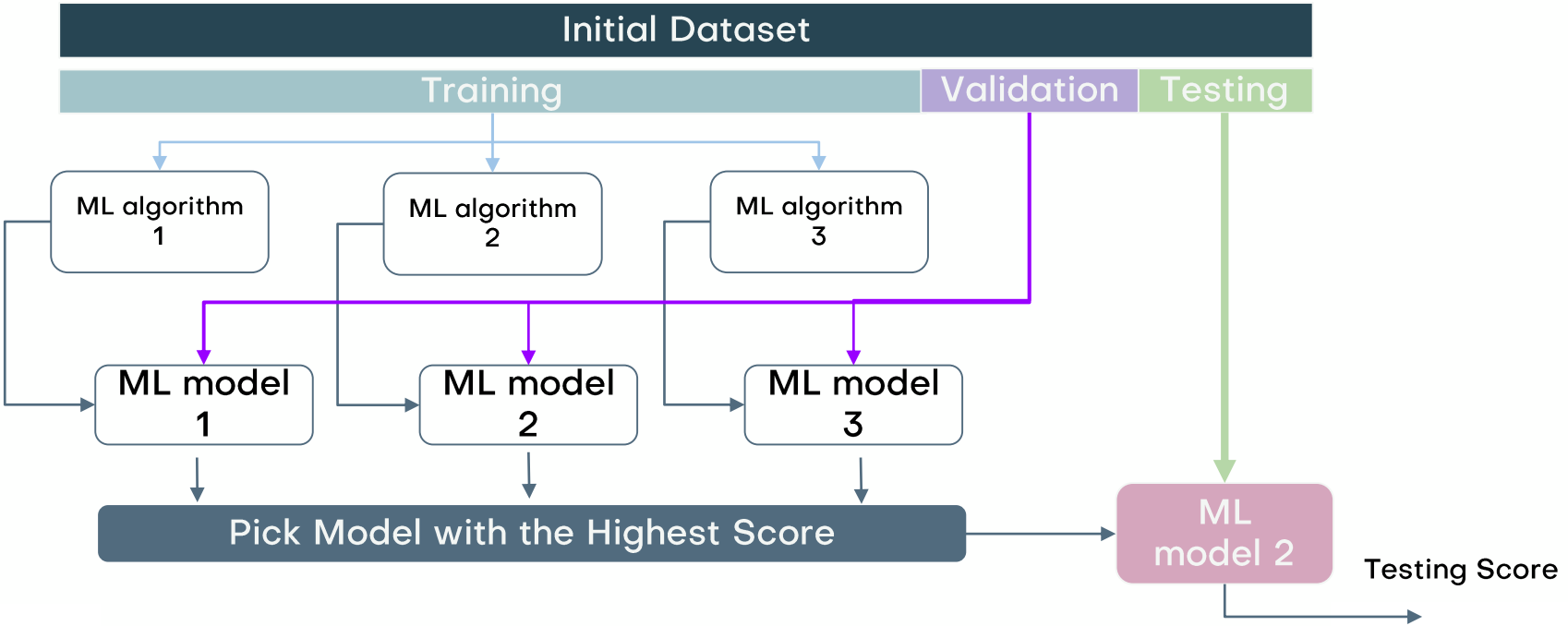
1. Hyperparameter Tuning

Hyperparameter Tuning

- It is the process of finding the best parameters for a learning algorithm.
- Examples of hyperparameters in Neural Networks:
  - Number of layers
  - Number of neurons inside of a layer
  - Learning rate
  - Regularization Factor

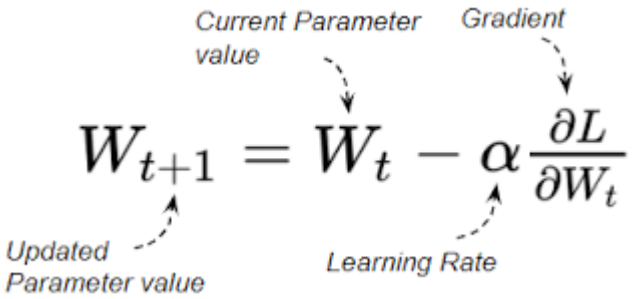


Hyperparameter Tuning



2. Varying Learning Rate

Learning Rate



Learning Rate

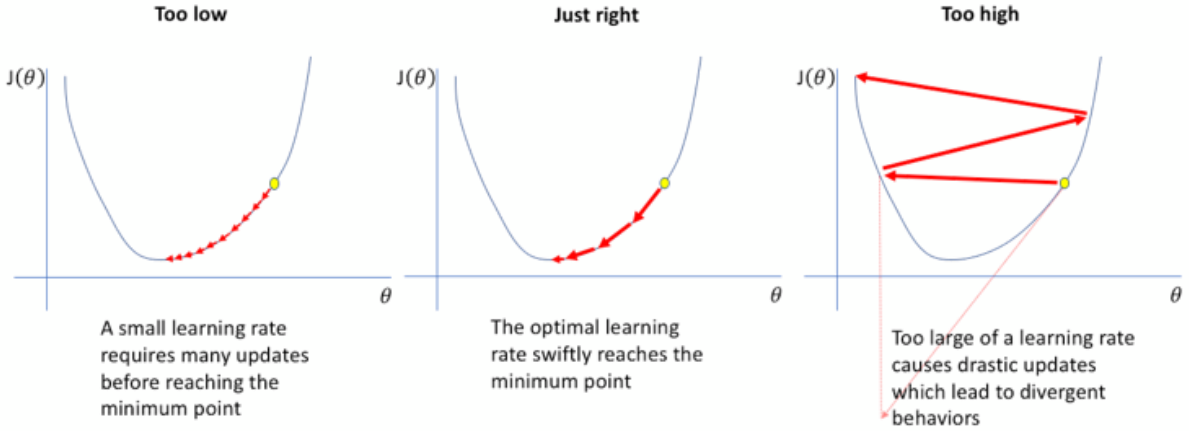
Current Parameter value

Gradient

$$W_{t+1} = W_t - \alpha \frac{\partial L}{\partial W_t}$$

Updated Parameter value

Learning Rate



Dynamic Learning Rate

Adapting the learning rate for your stochastic gradient descent optimization procedure can **increase performance** and **reduce training time!**

1. **Time-based Learning Rate Schedule**

$$LearningRate = LearningRate \times \frac{1}{1 + decay \times epoch}$$

2. **Drop-based Learning Rate Schedule**

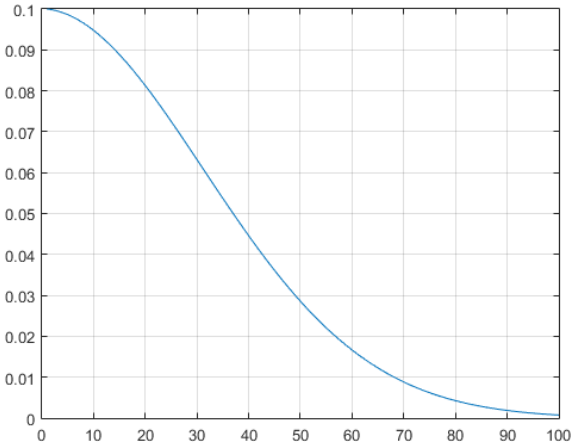
$$LearningRate = InitialLearningRate \times DropRate^{floor(\frac{1+epoch}{epochdrop})}$$

Dynamic Learning Rate

Decay argument

Example: LR = 0.1 / decay = 0.001

$$LearningRate = LearningRate \times \frac{1}{1 + decay \times epoch}$$



Dynamic Learning Rate

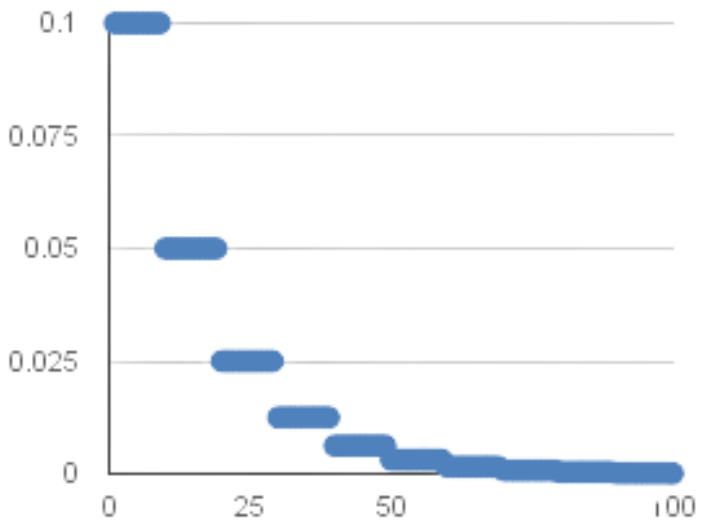
Systematically drop the learning rate at specific times during training.

LearningRateScheduler Callback

<u>Example:</u>

LR = 0.1 & drop it by a factor of 0.5

(half it) every 10 epochs



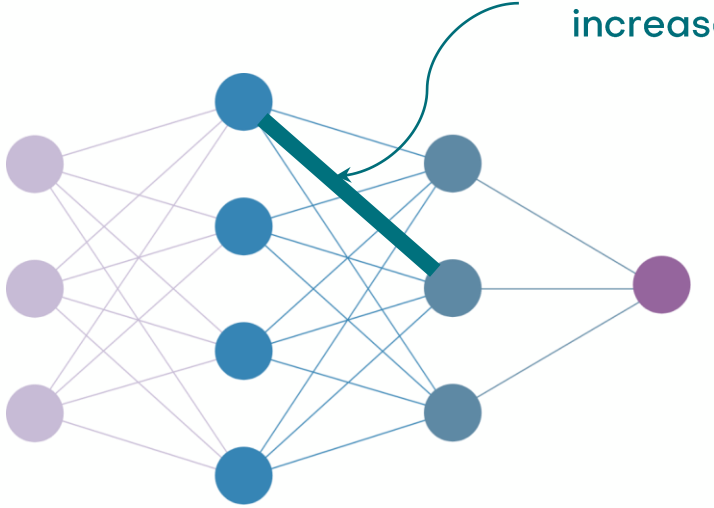
3. Batch Normalization

Batch Normalization

$$X\text{standardized} = \frac{x - mean}{std}$$

Training Data of m examples

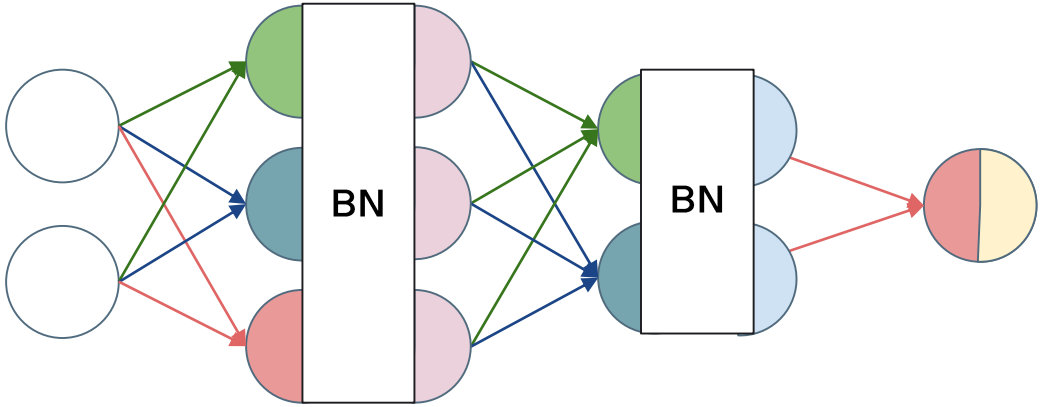
Standardize the data



What if this Weight increases during training?

Batch Normalization

Batch normalization standardizes the net outputs of a neuron when a batch is fed to the network.



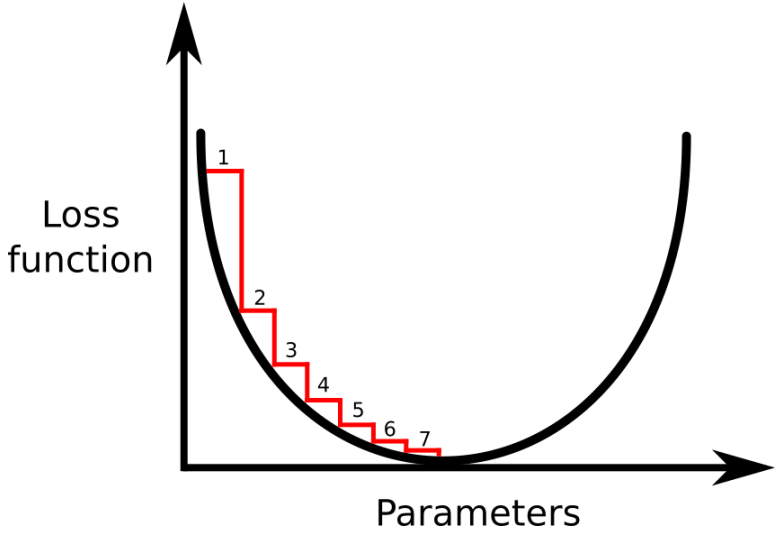
Improving DL Models: Optimization Methods

1. Gradient Descent Variants

Gradient Descent (Optimization) Variant

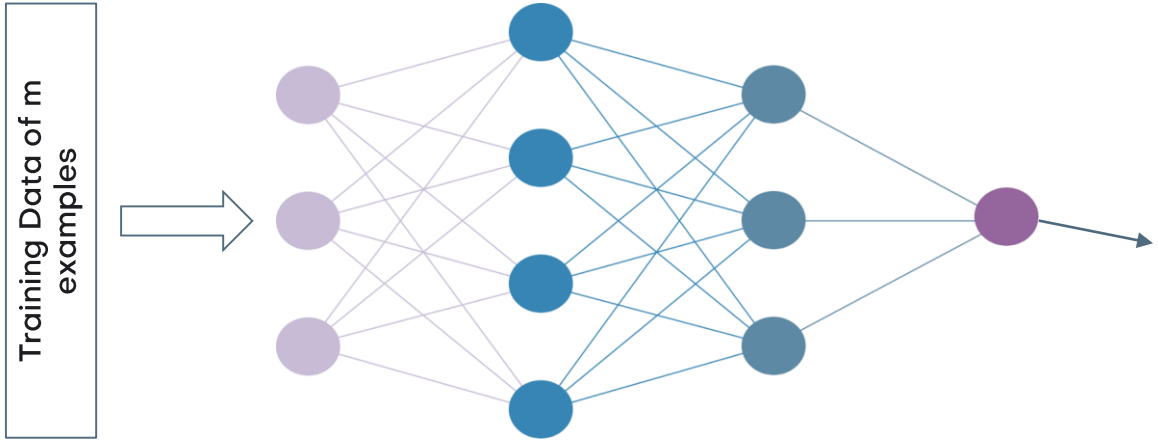
There are three variants of Gradient Descent:

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent



Batch Gradient Descent

- This method computes the gradient of the cost function using the entire training dataset.



Batch Gradient Descent

Advantages

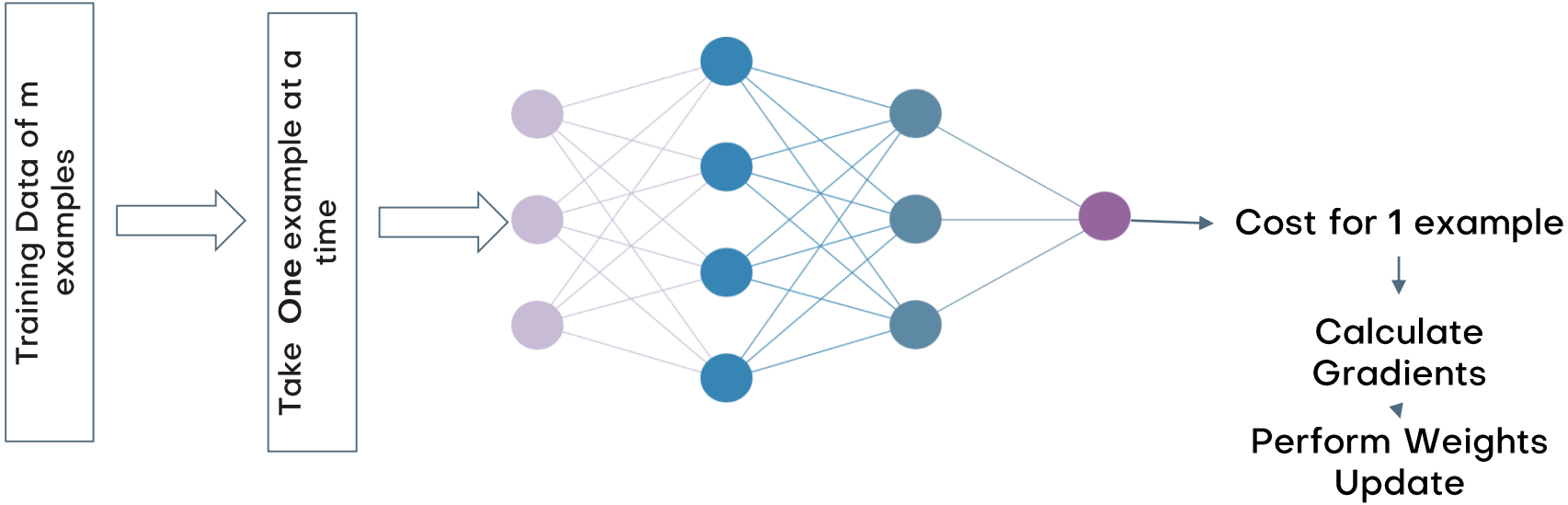
- It is guaranteed to converge to a minimum.

Disadvantages

- It can be very slow.
- It is intractable for datasets that do not fit in memory.
- It does not allow us to update our model online.

Stochastic Gradient Descent

- This method performs a parameter update for each training example $x(i)$ and label $y(i)$.



Stochastic Gradient Descent

Advantages

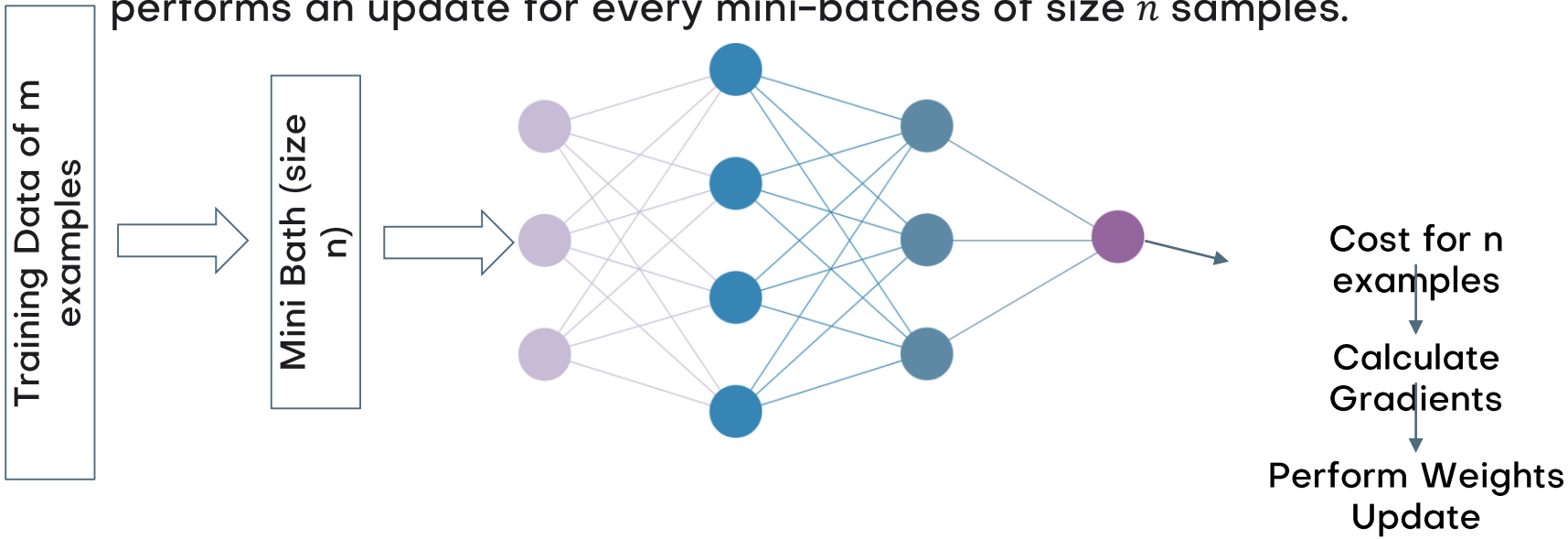
- It is usually much faster than batch gradient descent.
- It can be used to learn online.

Disadvantages

- It performs frequent updates with a high variance that cause the cost function to fluctuate heavily.

Mini-batch Gradient Descent

- This method combines the best of both batch and SGD and performs an update for every mini-batches of size n samples.



Mini-batch Gradient Descent

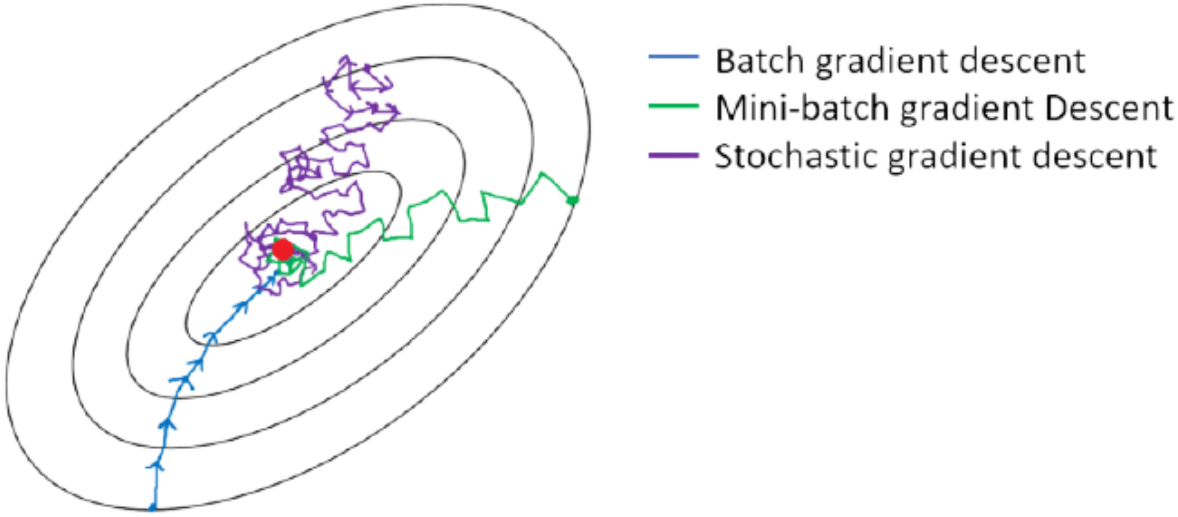
Advantages

- It reduces the variance of the parameter updates.
- It can lead to more stable convergence.

Disadvantages

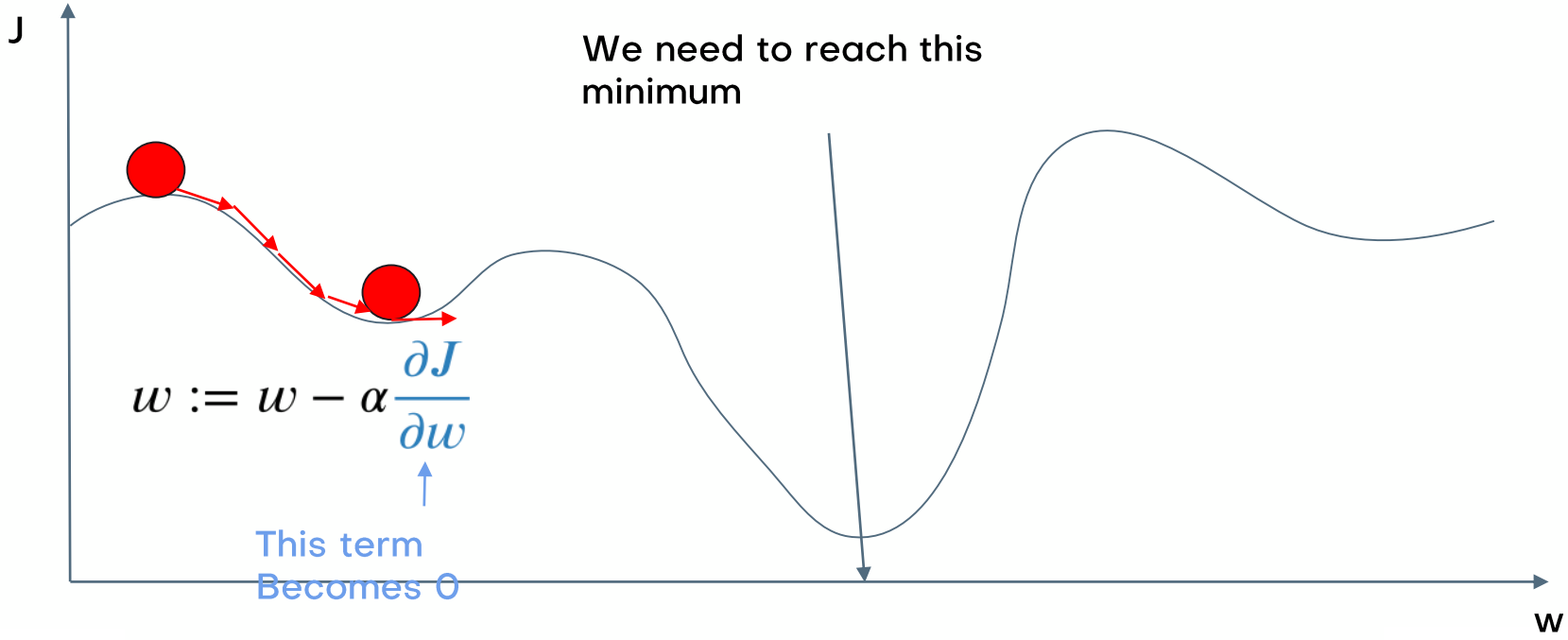
- We have to tune for the mini-batch size.

Comparison of the 3 techniques

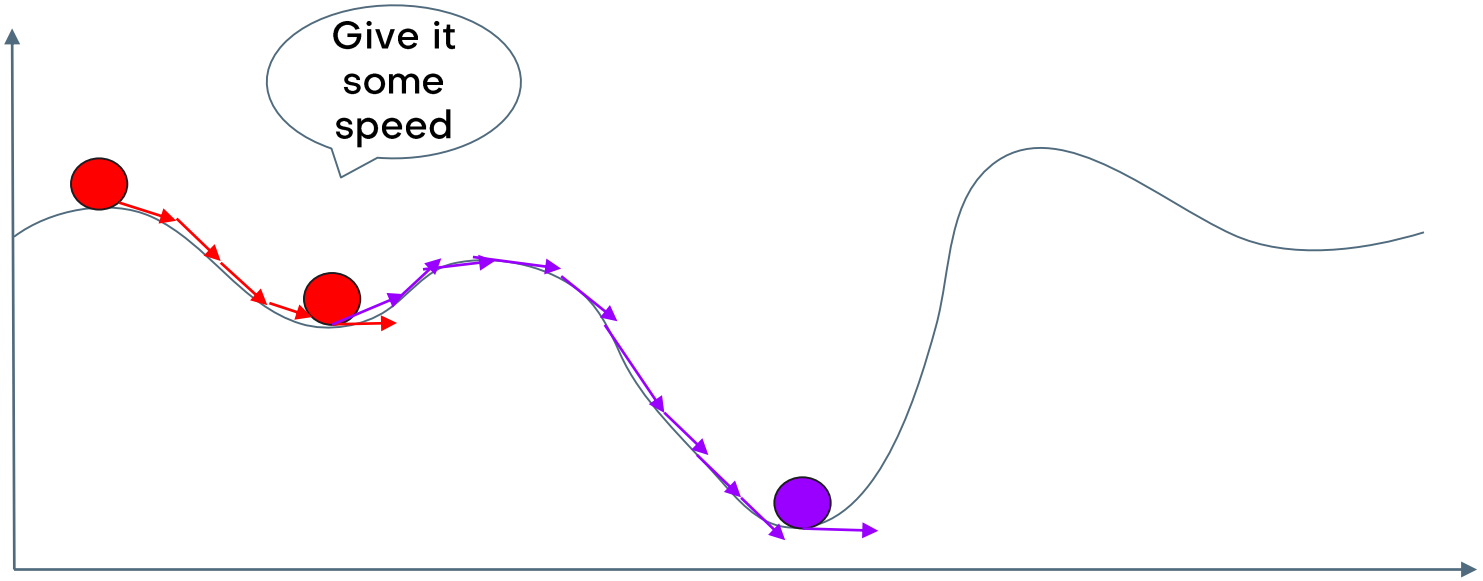


2. GD Optimization Algorithms

Gradient descent optimization algorithms



Gradient descent optimization algorithms



Gradient descent optimization algorithms

Gradient Descent Update Rule:

$$w := w - \alpha \frac{\partial J}{\partial w}$$

Other Algorithms Update Rule:

$$w := w - \alpha f\left(\frac{\partial J}{\partial w}\right)$$

Gradient Descent with Momentum

It's an algorithm that adds a fraction of the update vector at the previous time step to the current update vector.

Moving Average for the Gradients:

$$V_w := \beta V_w + (1 - \beta) \cdot \frac{\partial J}{\partial w}$$

Update Rule:

$$w := w - \alpha V_w$$

If this term becomes large, we might overshoot

RMS Prop

It's an algorithm developed in order to solve the momentum issue of the accumulating gradients.

Moving Average for the Square of the Gradients

$$S_w := \beta S_w + (1-\beta) \cdot \left(\frac{\partial J}{\partial w}\right)^2$$

Update Rule:

$$w := w - \alpha \cdot \frac{1}{\sqrt{S_w} + \epsilon}$$

If this term becomes large, the fraction will become close to 0, and the weight's won't update

Adam

Adam may be cited as the best optimizer since it combined the benefits of RMSProp and momentum.

$$V_w := \beta_1 V_w + (1 - \beta_1) \cdot \frac{\partial J}{\partial w} \qquad \hat{V}_w = \frac{V_w}{1 - \beta_1}$$

$$S_w := \beta_2 S_w + (1 - \beta_2) \cdot \left(\frac{\partial J}{\partial w} \right)^2 \qquad \hat{S}_w = \frac{S_w}{1 - \beta_2}$$

$$w := w - \frac{\alpha}{\sqrt{\hat{S}_w} + \epsilon} \hat{V}_w$$

Those 2 terms will increase together, so the fraction won't be 0 or infinite

3. Regularization

Regularization

- Neural Networks are an example of complex models.
- Regularization is used in Neural Networks to reduce their complexity.
- It does so by reducing the values of different weights.

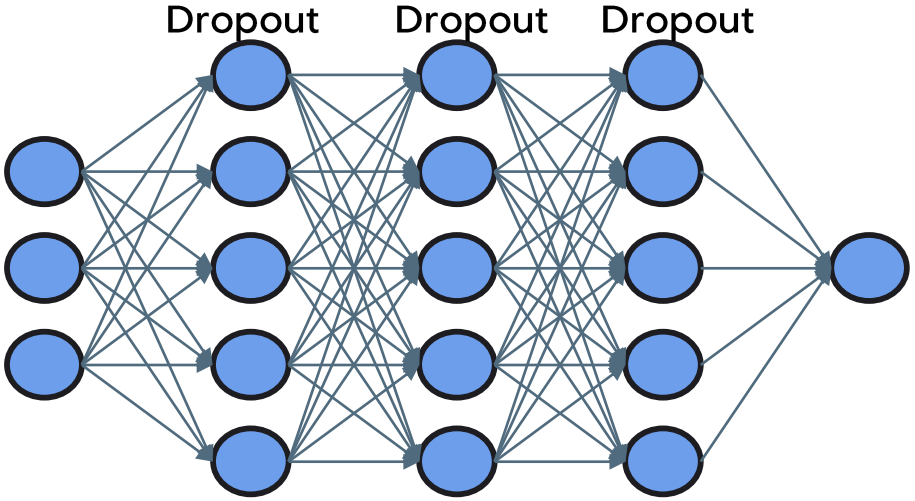
$$L2 - reg \longrightarrow New\ Cost = J + \frac{\lambda}{2m}\Sigma||w||^2$$

$$L1 - reg \longrightarrow New\ Cost = J + \frac{\lambda}{2m}\Sigma||w||$$

$$||w|| : Norm\ of\ a\ Weight\ Matrix$$

Dropout Regularization

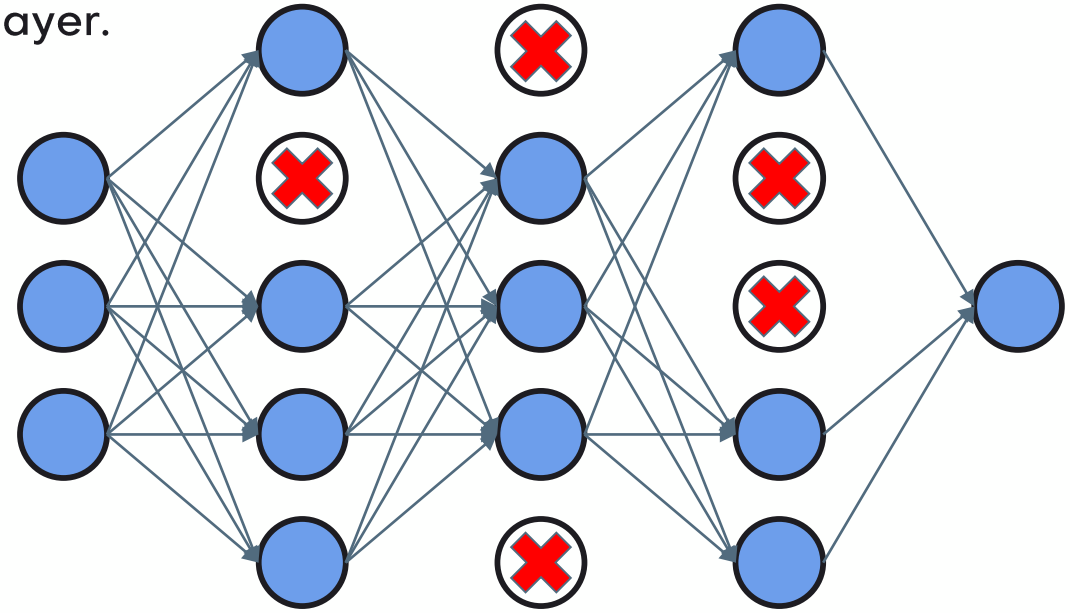
- At every iteration, it randomly removes specific neurons from a dropout layer.



Initial Network

Dropout

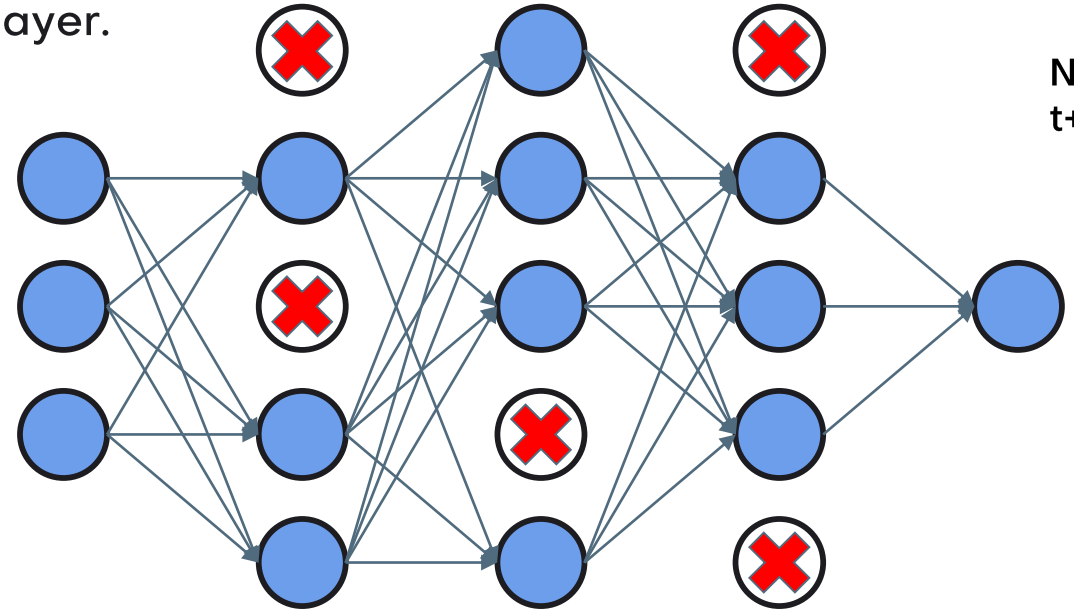
- At every iteration, it randomly removes specific neurons from a dropout layer.



Network at example t

Dropout

- At every iteration, it randomly removes specific neurons from a dropout layer.

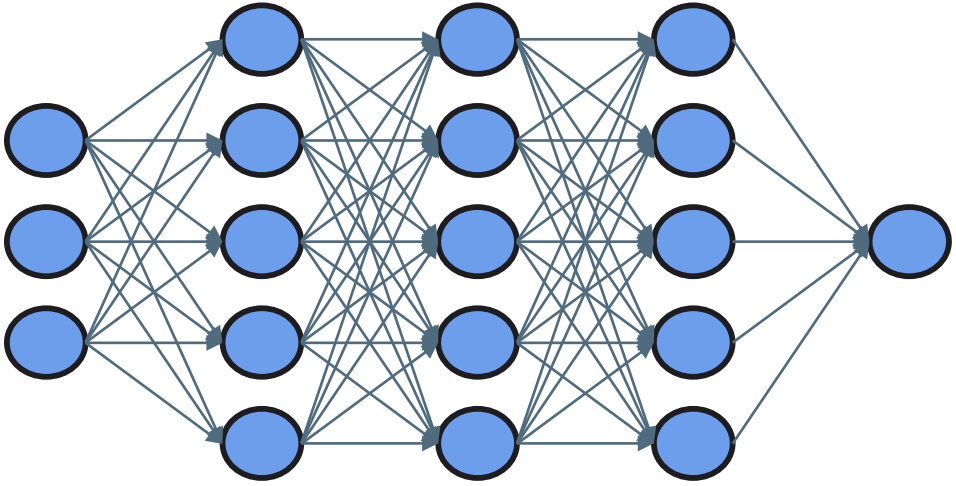


Network at example t+1

Dropout

To apply dropout, we need to:

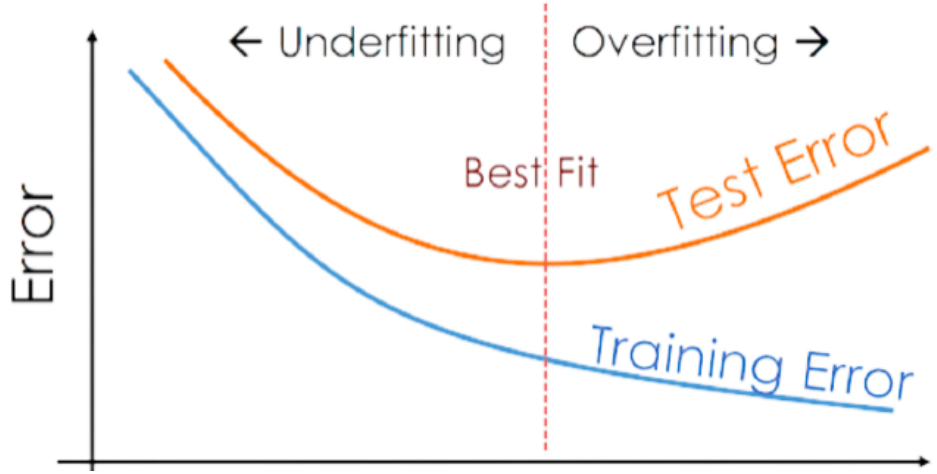
- Specify the layers where dropouts can occur
- Specify the dropout rate for this layer



4. Early Stopping

Early Stopping

- Early stopping is a technique allows us to stop training once the value being monitored (e.g validation loss or val_loss) has stopped improving.



Hands-on:
Iris Specie Detection

Alternative Framework: PyTorch

Overview

What is PyTorch?

- PyTorch is a library for Python
- Developed by Facebook's AI Research Lab
- Written in Python and C++
- Provides access to tools, libraries, and model architectures
- Provides pre-written tools used in CV, NLP…

Why PyTorch?

- Simple and easy to use
- Python usage
- Efficient memory usage
- Can scale to multiple GPUs
- Can be installed locally or in Google Colab

PyTorch Features

PyTorch provides two main features:

- Tensors that can run on GPUs unlike Numpy arrays
- Dynamic computational graph

PyTorch Features

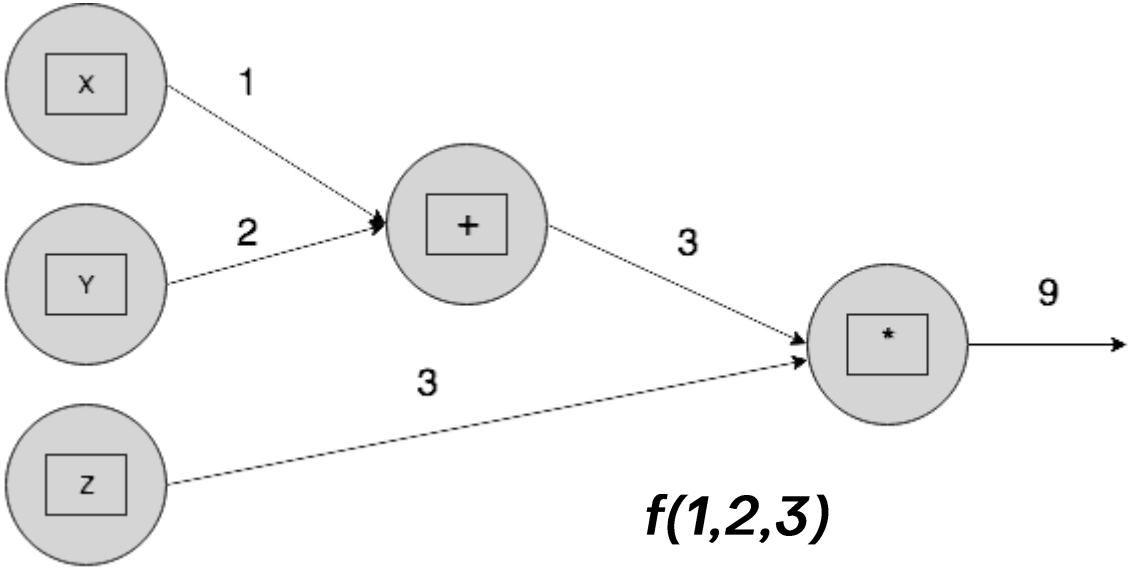
PyTorch provides two main features:

- **Tensors that can run on GPUs unlike Numpy arrays**
- Dynamic computational graph

PyTorch Features

PyTorch provides two main features:

- Tensors that can run on GPUs unlike Numpy arrays
- **Dynamic computational graph**

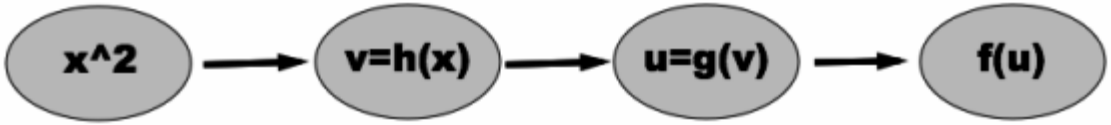


f(1,2,3)

PyTorch Features

PyTorch provides two main features:

- Tensors that can run on GPUs unlike Numpy arrays
- Dynamic computational graph



$$f(x)=ex$$
$$g(x)=\sin x$$
$$h(x)=x2$$
$$f(g(h(x)))=eg(h(x))$$

PyTorch Features

Computational graphs can be:

- Static: will not change once it is defined. Graph is set up then executed many times
- Dynamic: include the ability to adapt to a varying quantities in input data.

A graph is created on the fly

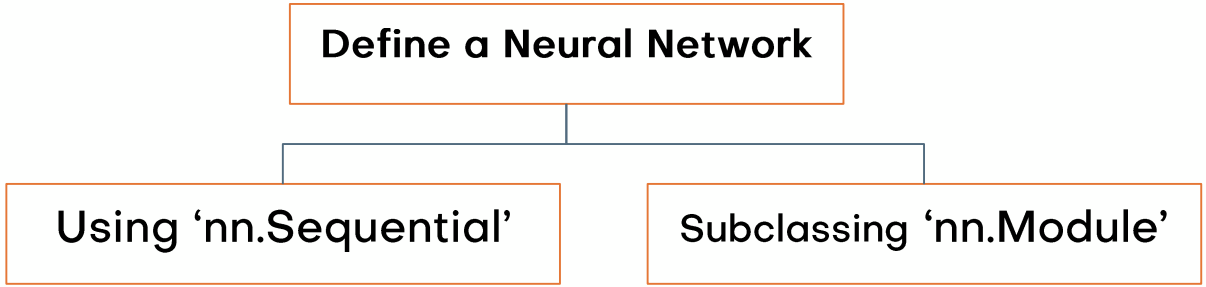

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

Creating Neural Networks using PyTorch

Neural Networks in PyTorch

- PyTorch provides the `torch.nn` module to define neural networks

```
import torch
import torch.nn as nn
```



Defining Neural Networks in PyTorch

Using ‘**nn.Sequential**’:

- nn.Sequential is a container
- Layers are stacked sequentially.
- Useful for networks where data flows linearly

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(in_features=64, out_features=128),
    nn.ReLU(),
    nn.Linear(128, 10),
    nn.Softmax(dim=1)
)
```

Defining Neural Networks in PyTorch

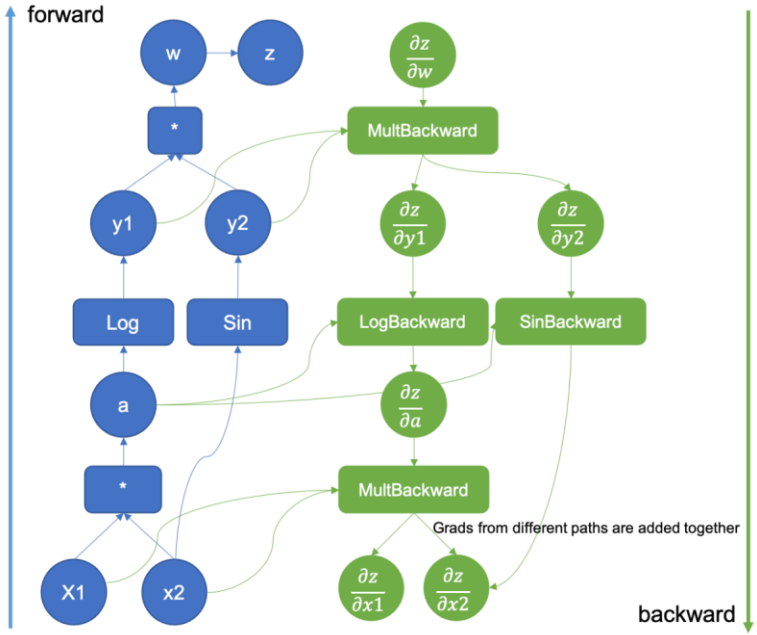
By subclassing '**nn.Module**':

- Better or complex architectures

```
import torch.nn as nn

class CustomNet(nn.Module):
    def __init__(self):
        super(CustomNet, self).__init__()
        self.layer1 = nn.Linear(64, 128)
        self.layer2 = nn.Linear(128, 10)

    def forward(self, x):
        x = nn.ReLU()(self.layer1(x))
        return nn.Softmax(dim=1)(self.layer2(x))

model = CustomNet()
```

Training Neural Networks using PyTorch

Training Neural Networks in PyTorch



Training Neural Networks in PyTorch

1. Define the network
2. Choose the loss function and optimizer
3. Training Loop:
    a. Feed data to the network (forward pass).
    b. Compute the loss using the chosen loss function.
    c. Zero out the gradients of the model using optimizer.zero_grad().
    d. Compute gradients with respect to the loss (backward pass) using loss.backward().
    e. Update the weights of the network using the chosen optimizer, e.g., optimizer.step().

Training Neural Networks in PyTorch

```
loss_fn = nn.BCELoss()  # binary cross entropy
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
n_epochs = 100
batch_size = 10

for epoch in range(n_epochs):
    for i in range(0, len(X), batch_size):
        Xbatch = X[i:i+batch_size]
        y_pred = model(Xbatch)
        ybatch = y[i:i+batch_size]
        loss = loss_fn(y_pred, ybatch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

شكراً لكم

Thank you

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority