NETZWERK ACADEMY

Deep Learning

1. Introduction

1.1. Definition

Deep Learning is a new area of Machine Learning, which has been introduced with the objective of moving Machine Learning closer to one of its original goals of Artificial Intelligence

1.2. Key Concepts

Deep-learning networks are distinguished from the more common single-hidden- layer neural networks by their depth

More than three layers (including input and output) qualifies as “deep” learning

In deep-learning networks, each layer of nodes trains on a distinct set of features based on the previous layer’s output

The further you advance into the neural net, the more complex the features your nodes can recognize, since they aggregate and recombine features from the previous layer

1.3. Types of Learning

Supervised: Learning with a labeled training set

Example: email *classification* with already labeled emails

Unsupervised: Discover patterns in unlabeled data

Example: *cluster* similar documents based on text

Reinforcement learning: learn to act based on feedback/reward

Example: learn to play Go, reward: *win or lose*

1.4. Benefits

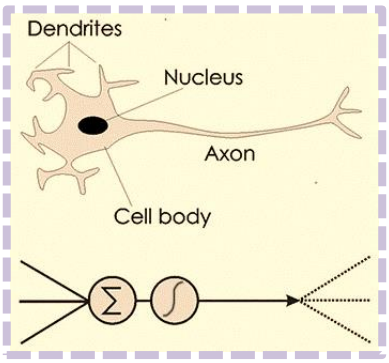
- Manually designed features are often over-specified, incomplete and take a long time to design and validate
- Learned Features are easy to adapt, fast to learn
- Deep learning provides a very flexible, (almost?) universal, learnable framework for representing world, visual and linguistic information.
- Can learn both unsupervised and supervised
- Effective end-to-end joint system learning
- Utilize large amounts of training data

1.5. Training

- Load Sample labeled data(batch)
- Forward it through the network, get predictions
- Back-propagate the errors
- Update the network weights
- Repeat the process

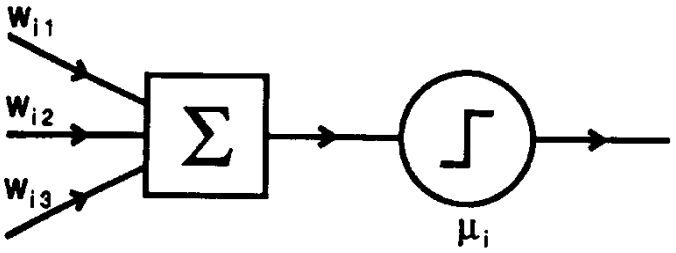
2. Neural Networks

2.1. Intuition



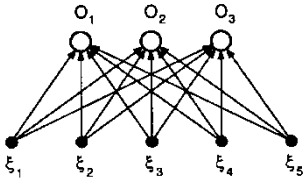
2.2. Neuron

Neuron is a mathematical function that models the functioning of a biological neuron. Typically, a neuron compute the weighted average of its input, and this sum is passed through a nonlinear function

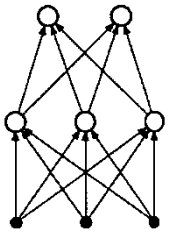


2.3. Perceptron

A perceptron is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data.
There are two types of Perceptron: Single layer and Multilayer.
Single layer Perceptron can learn only linearly separable patterns.
Multilayer Perceptron or feedforward neural networks with two or more layers have the greater processing power



Single-layer



Multilayer

2.4. Topologies

- Completely Connected
- Feed Forward
- Recurrent

3. Activation Functions

3.1. Introduction

Activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data. When comparing with a neuron-based model that is in our brains, the activation function is at the end deciding what is to be fired to the next neuron. Neural Network without non-linear activation functions will be just a simple linear regression model

3.2. Features & Problems

Vanishing Gradient problem: Neural Networks are trained using the process gradient descent. The gradient descent consists of the backward propagation step which is basically chain rule to get the change in weights in order to reduce the loss after every epoch

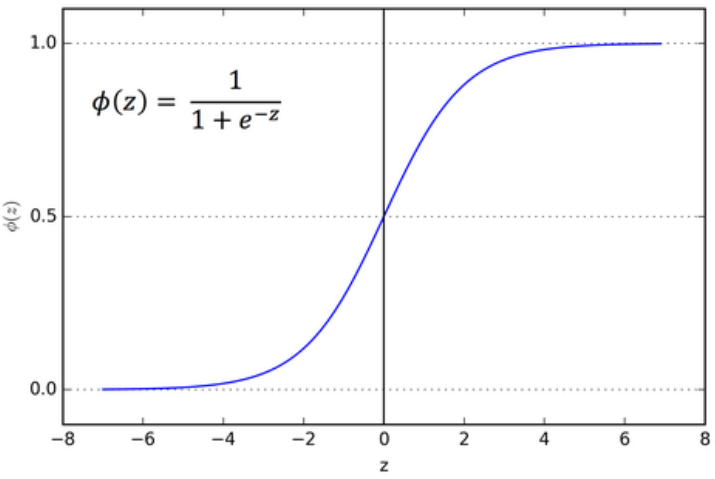
Zero-Centered: Output of the activation function should be symmetrical at zero so that the gradients do not shift to a particular direction

Computational Expense: Activation functions are applied after every layer and need to be calculated millions of times in deep networks. Hence, they should be computationally inexpensive to calculate.

Differentiable: As mentioned, neural networks are trained using the gradient descent process, hence the layers in the model need to differentiable or at least differentiable in parts. This is a necessary requirement for a function to work as activation function layer.

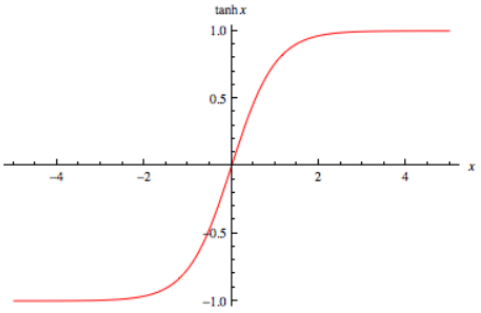
3.3. Sigmoid

The Sigmoid Function curve looks like an S-shape. It is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice



3.4. Tanh or hyperbolic tangent

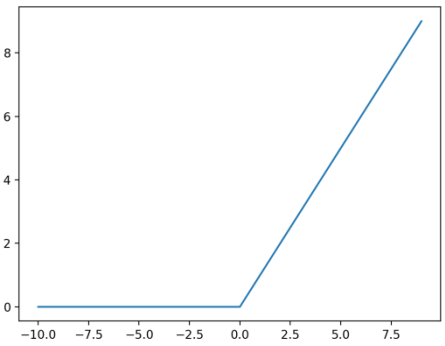
tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped). The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.



$$\tanh x = \frac{e^x - e^{-x}}{2} \div \frac{e^x + e^{-x}}{2} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

3.5. ReLU (Rectified Linear Unit) Activation Function

The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning

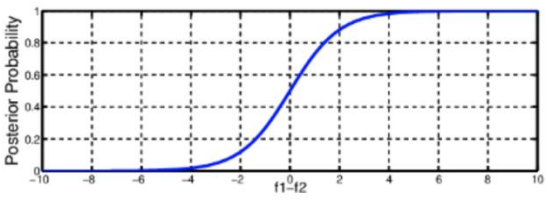


$$f(x) = x^+ = \max(0, x),$$

3.6. Softmax

Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector.

The softmax function is used as the activation function in the output layer of neural network models that predict a multinomial probability distribution.



$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

4. Artificial Neural Networks

4.1. Introduction

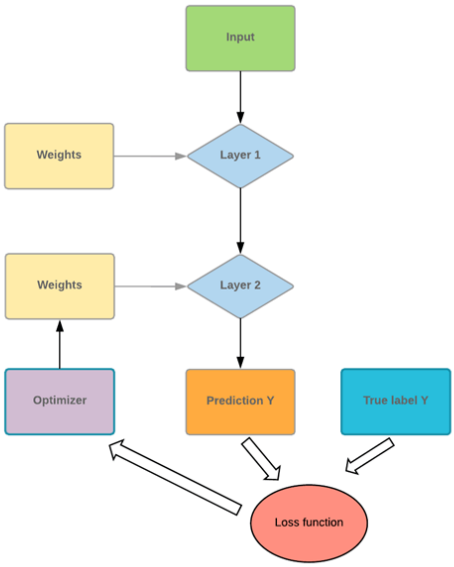
Artificial Neural Networks (ANN) is an Information processing paradigm inspired by biological nervous systems. ANN is composed of a system of neurons connected by synapses. ANN learn by example - Adjust synaptic connections between neurons. Combine speed of silicon with proven success of carbon → artificial brains

4.2. Neuron Model

- Neuron collects signals from dendrites
- Sends out spikes of electrical activity through an axon, which splits into thousands of branches.
- At end of each brand, a synapses converts activity into either exciting or inhibiting activity of a dendrite at another neuron.
- Neuron fires when exciting activity surpasses inhibitory activity`
- Learning changes the effectiveness of the synapses

4.3. Layers

A layer is where all the learning takes place. Inside a layer, there are an infinite amount of weights (neurons). A typical neural network is often processed by densely connected layers (also called fully connected layers). It means all the inputs are connected to the output.



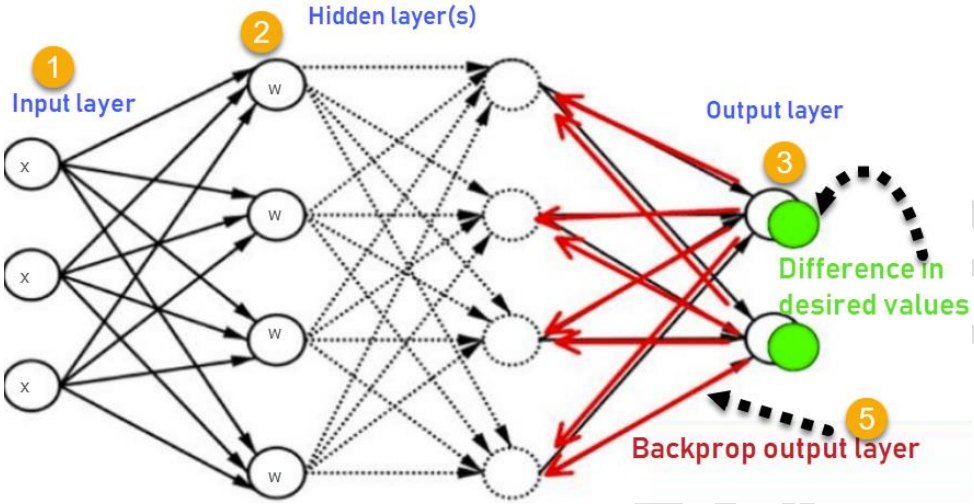
4.4. Limitations

Overfitting: A common problem with the complex neural net is the difficulties in generalizing unseen data. A neural network with lots of weights can identify specific details in the train set very well but often leads to overfitting.

Network size: A neural network with too many layers and hidden units are known to be highly sophisticated. A straightforward way to reduce the complexity of the model is to reduce its size. There is no best practice to define the number of layers. You need to start with a small amount of layer and increases its size until you find the model overfitting.

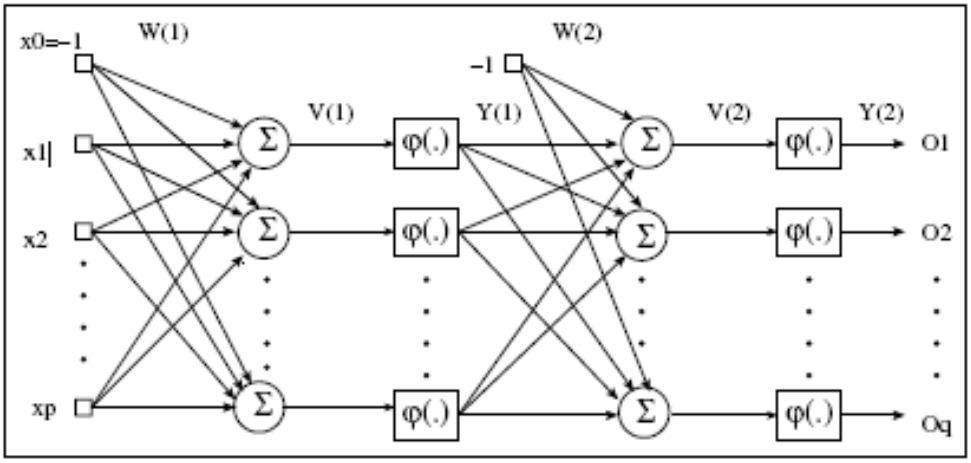
4.5. Back Propagation

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization.



4.6. Forward Propagation

Forward propagation calculates from input layer to output layer. It calculates weighted average of input and calculates activation function for each neuron



5. Cost Functions

5.1. Introduction

A cost function is a measure of error between what value your model predicts and what the value actually is. For example, say we wish to predict the value yi for data point xi.

$$\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$$

5.2. Types

Regression Loss Function: Regression models deals with predicting a continuous value for example given floor area, number of rooms, size of rooms, predict the price of the room. The loss function used in the regression problem is called “Regression Loss Function”. Below are the types under Regression

- Mean Squared Error Loss
- Mean Squared Logarithmic Error Loss
- Mean Absolute Error Loss

Binary Classification Loss Functions: Binary classification is a prediction algorithm where the output can be either one of two items, indicated by 0 or 1. The output of binary classification algorithms is a prediction score (mostly). So the classification happens based on the threshold the value (default value is 0.5). If the prediction score > threshold then 1 else 0.

- Binary Cross-Entropy
- Hinge Loss
- Squared Hinge Loss

Multi-class Classification Loss Functions: Multi-Class classification is those predictive modeling problems where there is more target variables/class. It is just the extension of binary classification problem.

- Multi-Class Cross-Entropy Loss
- Sparse Multiclass Cross-Entropy Loss
- Kullback Leibler Divergence Loss

5.3. Formulas

$$\text{MAE} = \underbrace{\frac{1}{n}}_{\text{test set}} \sum_{i=1}^{n} \underbrace{|y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i|}_{\text{actual value}} \qquad \text{MSE} = \underbrace{\frac{1}{n}}_{\text{test set}} \sum_{i=1}^{n} \underbrace{(y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i)^2}_{\text{actual value}}$$

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Predicted_i - Actual_i)^2}{N}} \qquad H(x) = \sum_{i=1}^{N} \underbrace{p(x)}_{\text{true label}} \log \underbrace{q(x)}_{\text{estimate}} \qquad L(y,\hat{y}) = -\sum_{j=0}^{M}\sum_{i=0}^{N}(y_{ij} * log(\hat{y}_{ij}))$$

BCE **CCE**

6. Optimization

6.1. Introduction

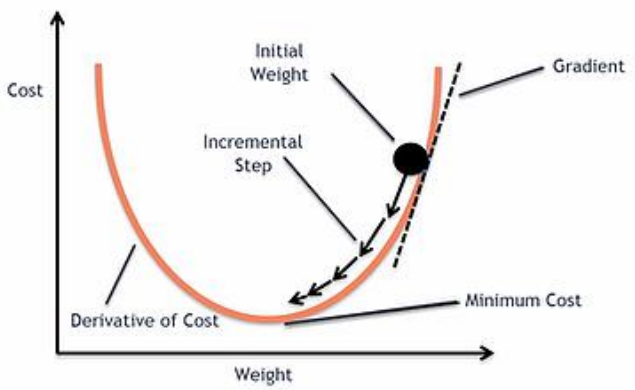
Optimization Algorithms are used to train a neural network model, to define a loss function in order to measure the difference between our model predictions and the label that we want to predict and look for certain set of weights, with which the neural network can make an accurate prediction, which automatically leads to a lower value of the loss function.

6.2. Gradient Descent

Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible

The mathematical method behind is called gradient descent

$$\theta_j \leftarrow \theta_j - \epsilon \nabla_{\theta_j} \mathcal{L}(\theta)$$



6.3. Stochastic Gradient Descent

SGD algorithm is an extension of the Gradient Descent and it overcomes some of the disadvantages of the GD algorithm. Gradient Descent has a disadvantage that it requires a lot of memory to load the entire dataset of n-points at a time to compute the derivative of the loss function. In the SGD algorithm derivative is computed taking one point at a time.

SGD performs a parameter update for *each* training example **x(i)** and label **y(i)**

$$\theta = \theta - \alpha \cdot \partial(J(\theta ; x(i), y(i))) / \partial \theta$$

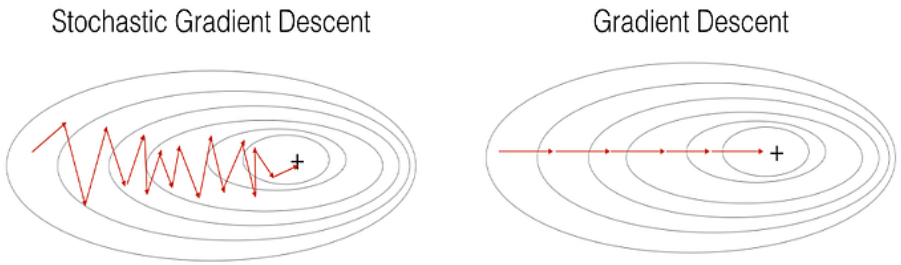
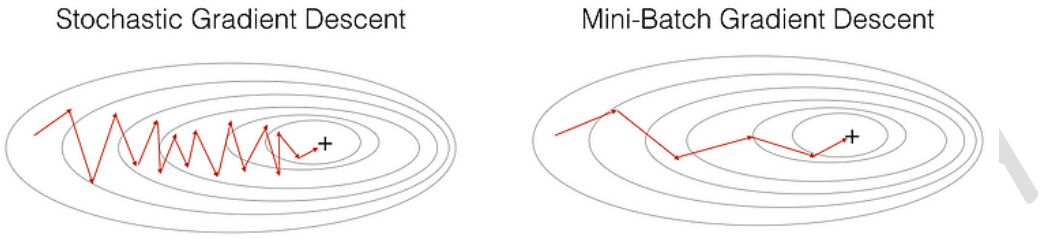


Figure 1 : SGD vs GD

"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

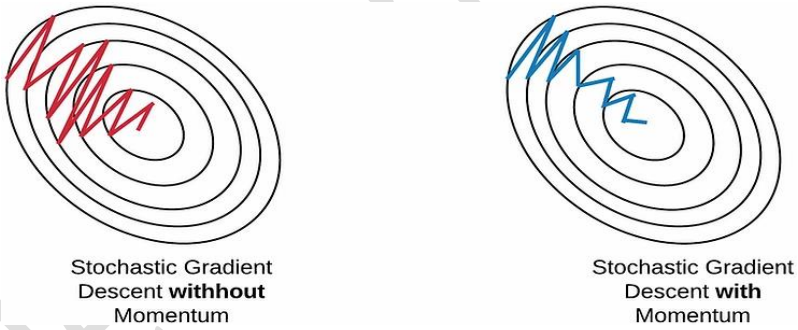
6.4. Mini Batch Stochastic Gradient Descent

MB-SGD algorithm is an extension of the SGD algorithm and it overcomes the problem of large time complexity in the case of the SGD algorithm. MB-SGD algorithm takes a batch of points or subset of points from the dataset to compute derivate. It is observed that the derivative of the loss function for MB-SGD is almost the same as a derivate of the loss function for GD after some number of iterations. But the number of iterations to achieve minima is large for MB-SGD compared to GD and the cost of computation is also large.



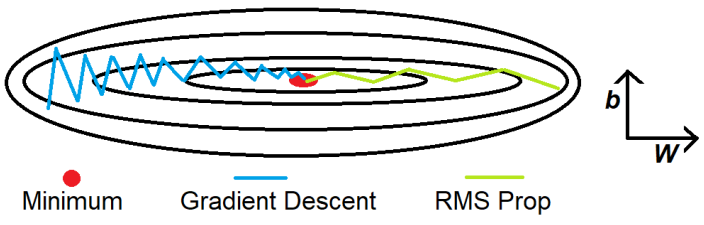
6.5. Stochastic Gradient Descent with Momemtum

A major disadvantage of the MB-SGD algorithm is that updates of weight are very noisy. SGD with momentum overcomes this disadvantage by denoising the gradients. Updates of weight are dependent on noisy derivative and if we somehow denoise the derivatives then converging time will decrease. The idea is to denoise derivative using exponential weighting average that is to give more weightage to recent updates compared to the previous update. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by 'γ'.



6.6. RMS Prop

RMS Prop stands for Root-Mean-Square Propagation. Similar to momentum, it is a technique to dampen out the motion in the y-axis. Let out try to understand how it works with the help of our previous example. For better understanding, let us denote the Y-axis as the bias *b* and the X-axis as the Weight *W*



In RMS prop, each update is done according to the equations described below. This update is done separately for each parameter.

$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \rho\nu_{t-1} + (1-\rho) * g_t^2$$
$$\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$\nu_t : Exponential\ Average\ of\ squares\ of\ gradients$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$

6.7. AdaM

AdaM stands for Adaptive Momentum. It combines the SGD with Momentum and RMS prop in a single approach making AdaM a very powerful and fast optimizer. We use error correction to solve the cold start problem in weighted average calculation, i.e., the first few values of weighted average are too far fro their real values. The V values incorporate the logic from Momentum, whereas S values incorporate the logic from RMS prop

$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \beta_1 * \nu_{t-1} - (1-\beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1-\beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta\frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$
$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$
$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$
$\beta_1, \beta_2 : Hyperparameters$

7. Convolution Neural Networks

7.1. Introduction

CNNs take a biological inspiration from the visual cortex. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. For example, some neurons fired when exposed to vertical edges and some when shown horizontal or diagonal edges. Having the neuronal cells in the visual cortex looking for specific characteristics is the basis behind CNNs

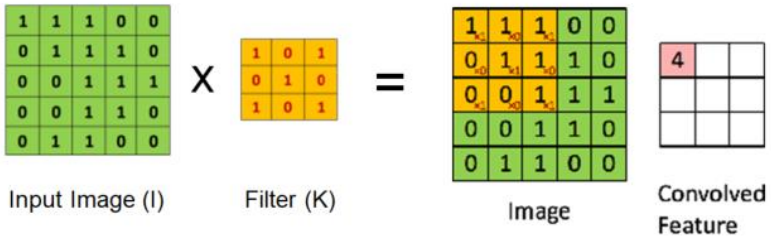
7.2. Layers

A Convolutional Neural Network (CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.

- **Convolutional Layer**

  The first layer of a Convolutional Neural Network is always a Convolutional Layer. Convolutional layers apply a convolution operation to the input, passing the result to the next layer. A convolution converts all the pixels in its receptive field into a single value. For example, if you would apply a convolution to an image, you will be decreasing the image size as well as bringing all the information in the field together into a single pixel. The final output of the convolutional layer is a vector. Based on the type of problem we need to solve and on the kind of features we are looking to learn, we can use different kinds of convolutions.

$$(I * K)_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} K_{ij} \cdot I_{x+i-1,y-j-1}$$



- **Padding**

  CNN learns the values of these filters on its own during the training process. Although we still need to specify parameters such as number of filters, filter size, padding, and stride before the training process.

  There are two types of results to the operation — one in which the convolved feature is reduced in dimensionality as compared to the input, and the other in which the dimensionality is either increased or remains the same

  This is done by applying **Valid Padding** in case of the former, or **Same Padding** in the case of the latter.

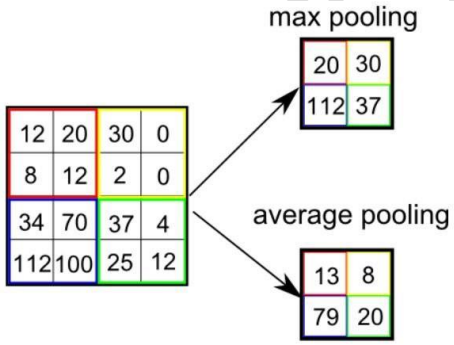
  When we augment the 5x5x1 image into a 6x6x1 image and then apply the 3x3x1 kernel over it, we find that the convolved matrix turns out to be of dimensions 5x5x1. Hence the name — **Same Padding**

if we perform the same operation without padding, we are presented with a matrix which has dimensions of the Kernel (3x3x1) itself — **Valid Padding**

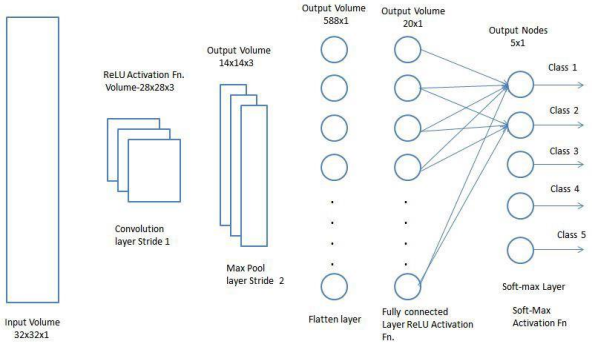
- **Activation Layer**
  An additional operation called Rectified Linear Unit (ReLU) has been used after every Convolution operation. The purpose of ReLU is to introduce non-linearity to the network

- **Pooling Layer**
  Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to **decrease the computational power required to process the data** by reducing the dimensions. There are two types of pooling average pooling and max pooling. Pooling layer **down samples** the volume spatially, **independently** in **each depth** slice of the input
  Max Pooling is to find the maximum value of a pixel from a portion of the image covered by the kernel. Max Pooling also performs as a **Noise Suppressant**. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction.
  Average Pooling returns the **average of all the values** from the portion of the image covered by the Kernel. Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism.

  Hence, we can say that **Max Pooling performs a lot better than Average Pooling**



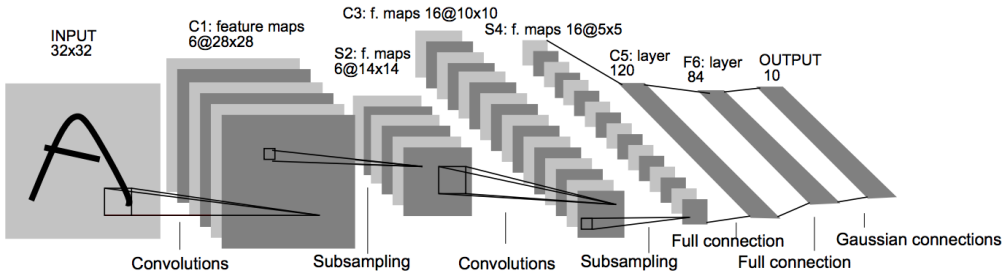
- **Fully Connected Layer**
  After flattening the final output and feed it to a regular Neural Network for classification purposes. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular neural networks



7.3. Architectures

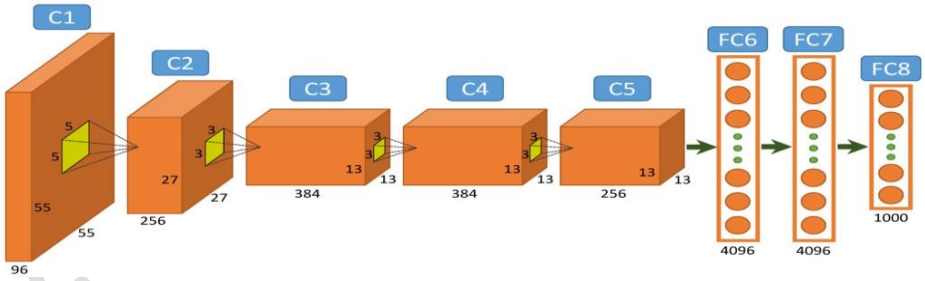
- **Lenet-5 Architecture**

  Yann Lecun's LeNet-5 model was developed in 1998 to identify handwritten digits for zip code recognition in the postal service. This pioneering model largely introduced the convolutional neural network. This architecture is an excellent "first architecture" for a CNN



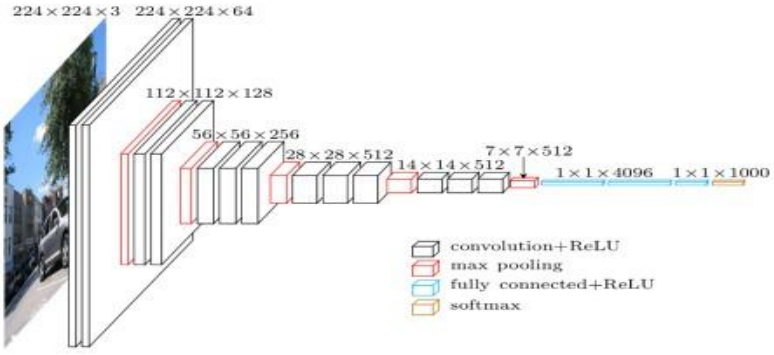
- **AlexNet Architecture**

  AlexNet was developed by Alex Krizhevsky et al. in 2012 to compete in the ImageNet competition. The general architecture is quite similar to LeNet-5, although this model is considerably larger. The success of this model (which took first place in the 2012 ImageNet competition) convinced a lot of the computer vision community to take a serious look at deep learning for computer vision tasks. Famous for winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012.



- **VGGNet Architecture**

  The VGG network, introduced in 2014, offers a deeper yet simpler variant of the convolutional structures discussed above. At the time of its introduction, this model was considered to be very deep.



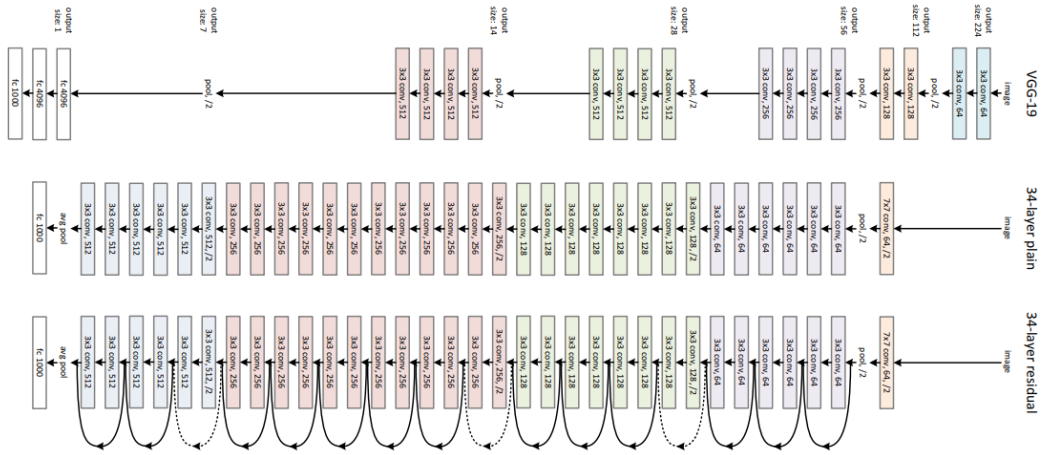
- **ResNet Architecture**

  ResNet, which was proposed in 2015 by researchers at Microsoft Research introduced a new architecture called Residual Network.

  Residual Block:

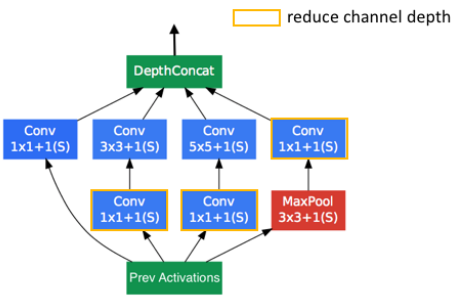
  In order to solve the problem of the vanishing/exploding gradient, this architecture introduced the concept called Residual Network. In this network we use a technique called *skip connections* . The skip connection skips training from a few layers and connects directly to the output.

  The approach behind this network is instead of layers learn the underlying mapping, we allow network fit the residual mapping. So, instead of say H(x), initial mapping, let the network fit, *F(x) := H(x) – x* which gives *H(x) := F(x) + x*.



- **Inception Architecture**

  In 2014, researchers at Google introduced the Inception network which took first place in the 2014 ImageNet competition for classification and detection challenges. The model is comprised of a basic unit referred to as an "Inception cell" in which we perform a series of convolutions at different scales and subsequently aggregate the results.



8. TensorFlow

8.1. Introduction

TensorFlow is a software library or framework, designed by the Google team to implement machine learning and deep learning concepts in the easiest manner. It combines the computational algebra of optimization techniques for easy calculation of many mathematical expressions. TensorFlow is also called a "Google" product. It includes a variety of machine learning and deep learning algorithms. TensorFlow can train and run deep neural networks for handwritten digit classification, image recognition, word embedding and creation of various sequence models.

8.2. Constructs

- Dataflow Graphs: entire computation
- Data Nodes: individual data or operations
- Edges: implicit dependencies between nodes
- Operations: any computation
- Constants: single values (tensors)
- All nodes return tensors, or higher-dimensional matrices
- How a node computes is indistinguishable to TensorFlow
- You are metaprogramming. No computation occurs yet!

8.3. Concepts

- Represents computations as graphs.
- Executes graphs in the context of Sessions.
- Represents data as tensors.
- Maintains state with Variables.
- Uses feeds and fetches to get data into and out of arbitrary operations.
- **Two Computation Phases**
    - **Construction phase**
        - **Assemble a graph**
        - ***C*reate a graph to represent and train a neural network**
    - **Execution phase**
        - **Use a session to execute ops in the graph**
        - **Repeatedly execute a set of training ops in the graph**

8.4. Tensors

Tensors are used as the basic data structures in TensorFlow language. Tensors represent the connecting edges in any flow diagram called the Data Flow Graph. Tensors are defined as multidimensional array or list.

Tensors are identified by the following three parameters −

Rank - Unit of dimensionality described within tensor is called rank. It identifies the number of dimensions of the tensor. A rank of a tensor can be described as the order or n-dimensions of a tensor defined.

Shape - The number of rows and columns together define the shape of Tensor.

Type - Type describes the data type assigned to Tensor's elements.

8.5. Variables

Variables are used to hold and update parameters, maintain state in the graph across calls to run(). They are In-memory buffers containing tensors. Must be explicitly initialized and can be saved to disk during and after training.

Class tf.Variable

8.6. Operations

Operations

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural-net building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

8.7. Modes

Eager mode: Eager execution is an imperative, define-by-run interface where operations are executed immediately as they are called from Python. This makes it easier toget started with TensorFlow, and can make research and development more intuitive.

Static mode: Predefine tensors and computation graphs then let TF engine to execute the graphs. Similar to defining Python functions.

8.8. Levels

- Primitive tensorflow: lowest, finest control and most flexible Suitable for most machine learning and deep learning algorithms.
- Keras(Mostly for deep learning):highest, most convenient to use, lack flexibility
- Tensorflow layers (Mostly for deep learning): somewhere at the middle.

8.9. Pipeline

- Define inputs and variable tensors(weights/parameters).
- Define computation graphs from inputs tensors to output tensors.
- Define loss function and optimizer
- Once the loss is defined, the optimizer will compute the gradient for you!
- Execute the graphs.

8.10. TensorBoard

TensorFlow includes a visualization tool, which is called the TensorBoard. It is used for analyzing Data Flow Graph and also used to understand machine-learning models. The important feature of TensorBoard includes a view of different types of statistics about the parameters and details of any graph in vertical alignment.

Deep neural network includes up to 36,000 nodes. TensorBoard helps in collapsing these nodes in high-level blocks and highlighting the identical structures

9. PyTorch

9.1. Introduction

PyTorch is defined as an open source machine/Deep learning library for Python. It is used for applications such as natural language processing. It is initially developed by Facebook artificial-intelligence research group, and Uber's Pyro software for probabilistic programming which is built on it. Originally, PyTorch was developed by Hugh Perkins as a Python wrapper for the LusJIT based on Torch framework. PyTorch redesigns and implements Torch in Python while sharing the same core C libraries for the backend code. PyTorch developers tuned this back-end code to run Python efficiently. They also kept the GPU based hardware acceleration as well as the extensibility features.

9.2. Features

- Easy Interface − easy to use API. The code execution in this framework is quite easy. Also need a fewer lines to code in comparison.
- It is easy to debug and understand the code.
- Python usage − This library is considered to be Pythonic which smoothly integrates with the Python data science stack.
- It can be considered as NumPy extension to GPUs.
- Computational graphs- PyTorch provides an excellent platform for dynamic computational graphs. Thus a user can change them during runtime.
- It includes many layers as Torch.
- It includes lot of loss functions.
- It allows building networks whose structure is dependent on computation itself.
- NLP: account for variable length sentences. Instead of padding the sentence to a fixed length, we create graphs with different number of LSTM cells based on the sentence's length.

9.3. Tensors

Pytorch tensors are just like numpy arrays but they can run on GPU

Normal Tensor: dtype = torch.FloatTensor
GPU Tensor: dtype = torch.cuda. FloatTensor
Cuda stands for common unified device architecture

9.4. Variable

A pytorch variable is a node in computational graph where variable.data is tensor

9.5. Model

In PyTorch, a model is represented by a regular Python class that inherits from the Module class.

- **Two components**
  - **__init__(self): it defines the parts that make up the model —in our case, two *parameters*, *a* and *b***
  - **forward(self,x): it performs the actual computation, that is, it outputs a prediction, given the input x**

9.6. Dataset

In PyTorch, a dataset is represented by a regular Python class that inherits from the Dataset class. You can think of it as a kind of a Python list of tuples, each tuple corresponding to one point (features, label). Unless the dataset is huge (cannot fit in memory), you don't explictly need to define this class. Use TensorDataset

Ex:

```
from torch.utils.data import Dataset, TensorDataset

class CustomDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)

x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

train_data = CustomDataset(x_train_tensor, y_train_tensor)
print(train_data[0])

train_data = TensorDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

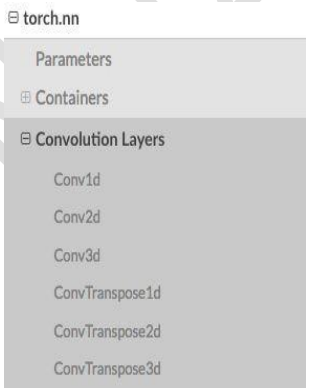
9.7. Dataloader

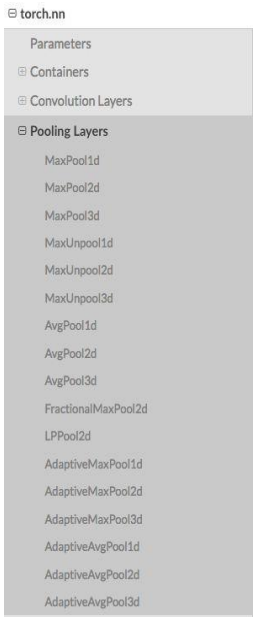
Use PyTorch's Dataloader class with which dataset to use, the desired mini-batch size and if we'd like to shuffle it or not. Lloader will behave like an iterator, so we can loop over it and fetch a different mini-batch every time.

```
from torch.utils.data import DataLoader

train_loader = DataLoader(dataset=train_data, batch_size=16, shuffle=True)
```

9.8. Layers



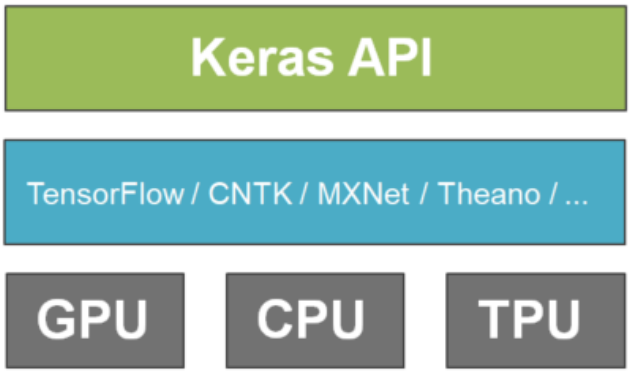


10. Keras

10.1. Introduction

Keras runs on top of open source machine libraries like TensorFlow, Theano or Cognitive Toolkit (CNTK). Theano is a python library used for fast numerical computation tasks. TensorFlow is the most famous symbolic math library used for creating neural networks and deep learning models. TensorFlow is very flexible and the primary benefit is distributed computing. CNTK is deep learning framework developed by Microsoft. It uses libraries such as Python, C#, C++ or standalone machine learning toolkits.

Theano and TensorFlow are very powerful libraries but difficult to understand for creating neural networks. Keras is based on minimal structure that provides a clean and easy way to create deep learning models based on TensorFlow or Theano. Keras is designed to quickly define deep learning models. Well, Keras is an optimal choice for deep learning applications.



10.2. Features and Benefits

- Consistent, simple and extensible API.
- Minimal structure - easy to achieve the result without any frills.
- It supports multiple platforms and backends.
- It is user friendly framework which runs on both CPU and GPU.
- Highly scalability of computation.
- Keras is an API for specifying & training differentiable programs
- Keras is the official high-level API of Tensor Flow
- Better optimized for TF and integration with TF-specific features
- A focus on user experience.
- Large adoption in the industry and research community.
- Multi-backend, multi-platform and Easy productization of models
- Keras neural networks are written in Python which makes things simpler.
- Keras supports both convolution and recurrent networks.
- Deep learning models are discrete components, so that, you can combine into many ways

10.3. Modules

Initializers − Provides a list of initializers function during model creation phase of machine/deep learning.

Regularizers − Provides a list of regularizers function.

Constraints − Provides a list of constraints function.
Activations − Provides a list of activator function.
Losses − Provides a list of loss function.
Metrics − Provides a list of metrics function.
Optimizers − Provides a list of optimizer function.
Callback − Provides a list of callback function. We can use it during the training process to print the intermediate data as well as to stop the training itself (EarlyStopping method) based on some condition.
Text processing − Provides functions to convert text into NumPy array suitable for machine learning.
Image processing − Provides functions to convert images into NumPy array suitable for Deep learning.
Sequence processing − Provides functions to generate time based data from the given input data.
Backend − Provides function of the backend library like TensorFlow and Theano.
Utilities − Provides lot of utility function useful in deep learning

10.4. Layers

Keras layers are the primary building block of Keras models. Each layer receives input information, do some computation and finally output the transformed information. The output of one layer will flow into the next layer as its input.
Dense layer is the regular deeply connected neural network layer.
Dropout is one of the important concept in the Deep learning.
Flatten is used to flatten the input.
Reshape is used to change the shape of the input.
Permute is also used to change the shape of the input using pattern.
RepeatVector is used to repeat the input for set number, n of times.
Keras contains a lot of layers for creating Convolution based ANN, popularly called as ***Convolution Neural Network (CNN)***.
Pooling Layer is used to perform max pooling operations on temporal data.
Locally connected layers are similar to Conv1D layer but the difference is Conv1D layer weights are shared but here weights are unshared.
Merge Layer is used to merge a list of inputs.
Embedding Layer performs embedding operations in input layer.

10.5. Three API Styles

- **The Sequential Model**
  - Dead simple
  - Only for single-input, single-output, sequential layer stacks
  - Good for 70+% of use cases

- **The functional API**
  - Like playing with Lego bricks
  - Multi-input, multi-output, arbitrary static graph topologies
  - Good for 95% of use cases

- **Model sub classing**
  - Maximum flexibility
  - Larger potential error surface

10.6. Models

Model groups layers into an object with training and inference features and returns a layer instance.

Arguments

Inputs: The input(s) of the model: a Keras Input object or list of Keras Input objects.

Outputs: The output(s) of the model.

Name: String, the name of the model.

Syntax: tf.keras.Model()

10.7. Dense Layer

The dense layer is a neural network layer that is connected deeply, which means each neuron in the dense layer receives input from all neurons of its previous layer. The dense layer is found to be the most commonly used layer in the models.

Syntax

keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

Keras Dense Layer Operation

The dense layer function of Keras implements following operation –

output = activation(dot(input, kernel) + bias)

10.8. Callback

A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc).

You can use callbacks to:

- Write TensorBoard logs after every batch of training to monitor your metrics
- Periodically save your model to disk
- Do early stopping
- Get a view on internal states and statistics of a model during training

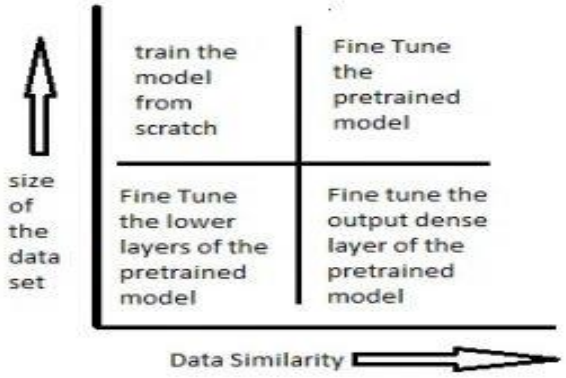
```
my_callbacks = [
  tf.keras.callbacks.EarlyStopping(patience=2),
  tf.keras.callbacks.ModelCheckpoint(filepath='model.{epoch:02d}-
{val_loss:.2f}.h5'),
  tf.keras.callbacks.TensorBoard(log_dir='./logs'),
]
model.fit(dataset, epochs=10, callbacks=my_callbacks)
```

10.9. Transfer Learning

Storing knowledge gained while solving one problem, and applying it to a different but related problem.

Why use Transfer Learning.

- Training a Model, from scratch, requires lot of input data.
- Very deep networks are very resource and cash expensive.
- Determining the topology, and hyper parameters is black magic.



10.10. Fine Tune Predefined a Model

- **New dataset is small and similar to original dataset.**
  Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns.
  Since the data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well.
  Hence, the best idea might be to train a linear classifier on the CNN codes.

- **New dataset is large and similar to the original dataset.**
  Since we have more data, we can have more confidence that we won't overfit if we were to try to fine-tune through the full network.

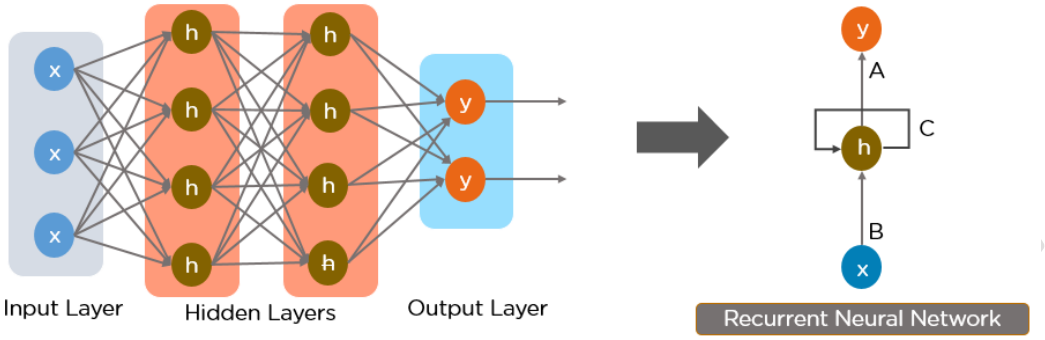
- **New dataset is small but very different from the original dataset.**
  Since the data is small, it is likely best to only train a linear classifier.
  Since the dataset is very different, it might not be best to train the classifier form the top of the network, which contains more dataset-specific features.
  Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.

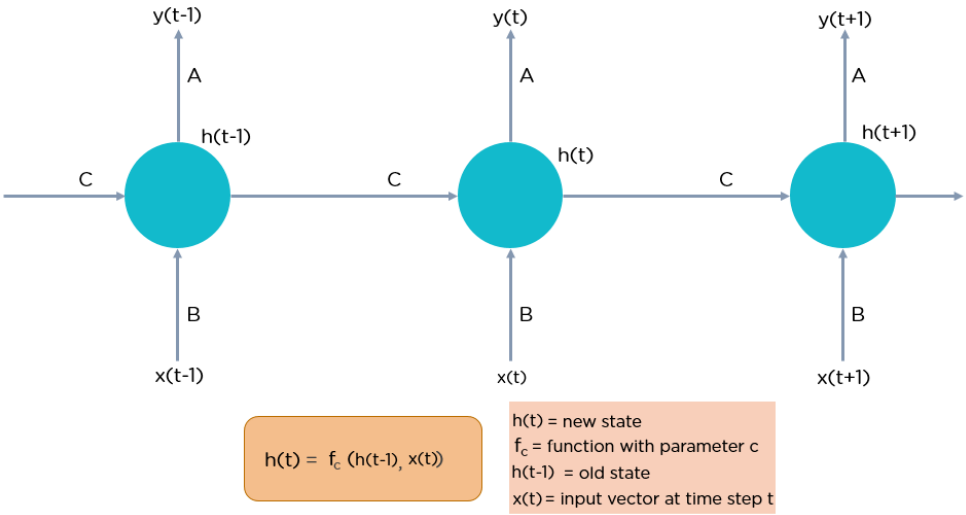
- **New dataset is large and very different from the original dataset.**
  Since the dataset is very large, we can afford to train a ConvNet from scratch.
  However, in practice it is very often still beneficial to initialize with weights from a pretrained model. In this case, we would have enough data and confidence to fine-tune through the entire network.

11. Recurrent Neural Networks

11.1. Introduction

Recurrent Neural Network works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer. The nodes in different layers of the neural network are compressed to form a single layer of recurrent neural networks. A, B, and C are the parameters of the network





11.2. Neuron

In the structure, the blue RNN block, applies something called as a recurrence formula to the input vector and also its previous state. In any case, the First letter has nothing preceding it, let's take the Second letter, so at the time this letter is supplied to the network, a recurrence formula is applied to the letter and the previous state which is the first letter. These are known as various time steps of the input. So if at time t, the input is second letter, at time t-1, the input was first letter. The recurrence formula is applied to both and we get a new state.

The formula for the current state can be written as below. Ht is the new state, ht-1 is the previous state while xt is the current input

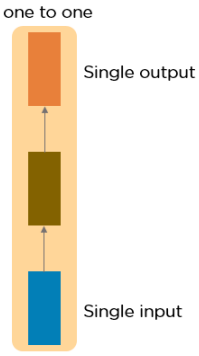
$$h_t = f(h_{t-1}, x_t)$$

11.3. Types

There are four types of Recurrent Neural Networks:

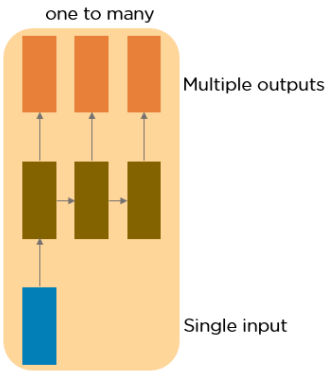
- **One to One**

  This type of neural network is known as the Vanilla Neural Network. It's used for general machine learning problems, which has a single input and a single output.



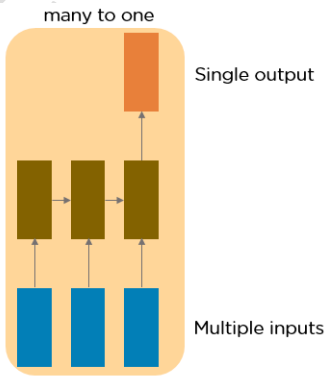
- **One to Many**

  This type of neural network has a single input and multiple outputs. An example of this is the image caption.

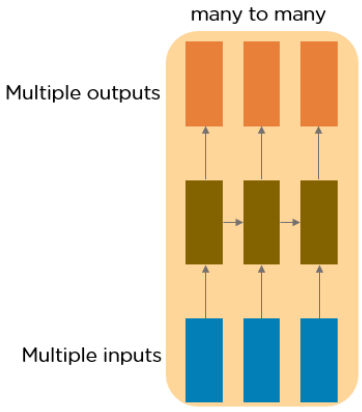


- **Many to One**

  This RNN takes a sequence of inputs and generates a single output. Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments.

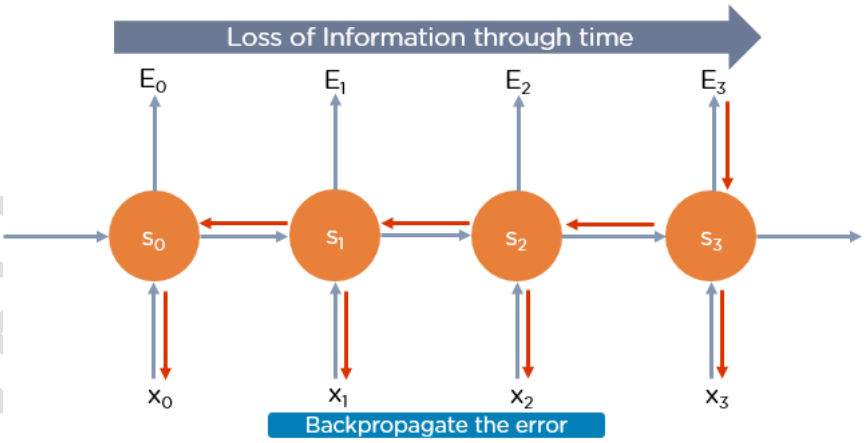


- **Many to Many**
  This RNN takes a sequence of inputs and generates a sequence of outputs. Machine translation is one of the examples



11.4. Problems

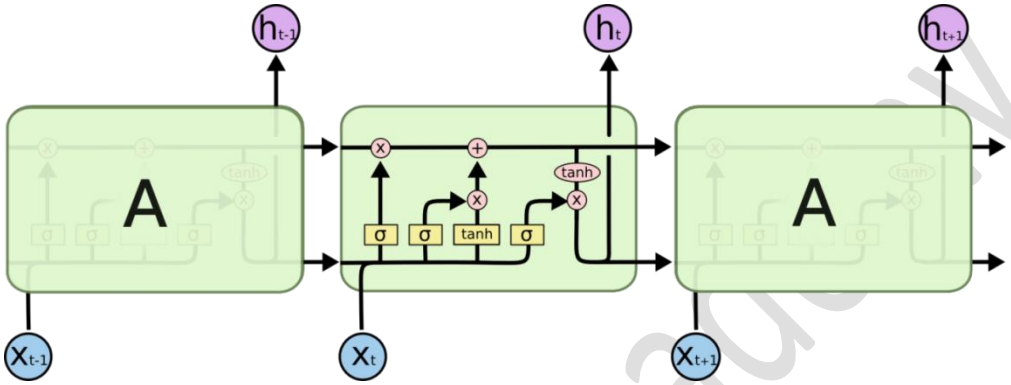
- **Vanishing Gradient Problem**
  Recurrent Neural Networks enable you to model time-dependent and sequential data problems, such as stock market prediction, machine translation, and text generation. You will find, however, that recurrent Neural Networks are hard to train because of the gradient problem. RNNs suffer from the problem of vanishing gradients. The gradients carry information used in the RNN, and when the gradient becomes too small, the parameter updates become insignificant. This makes the learning of long data sequences difficult.



- **Exploding Gradient Problem**
  While training a neural network, if the slope tends to grow exponentially instead of decaying, this is called an Exploding Gradient. This problem arises when large error gradients accumulate, resulting in very large updates to the neural network model weights during the training process. Long training time, poor performance, and bad accuracy are the major issues in gradient problems.

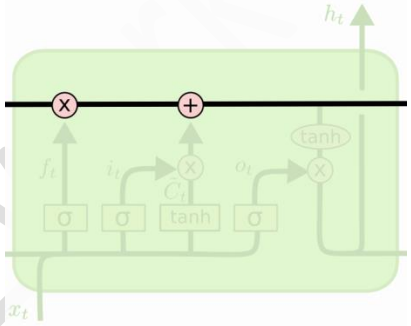
11.5. LSTM – Long Short Term Memory

LSTM is a special neuron for memorizing long-term dependencies. LSTM contains an internal state variable which is passed from one cell to the other and modified by Operation Gates. LSTMs are explicitly designed to avoid the long-term dependency problem.
LSTM is smart enough to determine how long to hold onto old information, when to remember and forget, and how to make connections between old memory with the new input. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



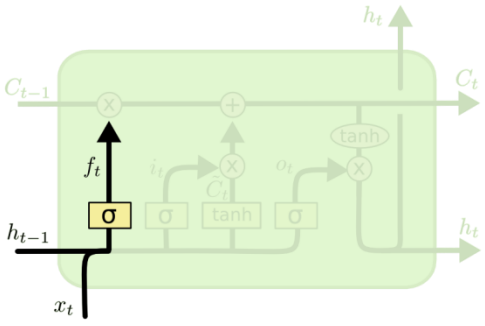
11.6. Breakdown

- **Hidden states**
  - Carry previous information
  - Build a direct path

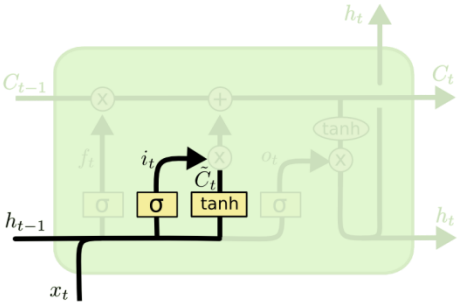


- **Input gates**

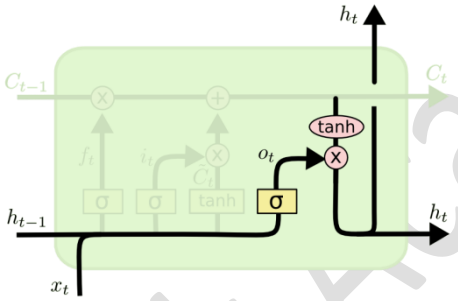
  What is kept from previous state



- **Forget gates**
  What should be added to hidden state



- **Output gates**
  What is reported as output



11.7. Steps

- LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at ht−1 and xt, and outputs a number between 0 and 1 for each number in the cell state Ct−1. 1 represents "completely keep this" while a 0 represents "completely get rid of this."
- sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C~t, that could be added to the state. In the next step, we'll combine these two to create an update to the state.
- This step updates the old cell state, Ct−1, into the new cell state Ct. The previous steps already decided what to do, we just need to actually do it.
- We multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add [it∗C~t]. This is the new candidate values, scaled by how much we decided to update each state value.

- Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanhtanh (to push the values to be between −1−1 and 11) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.