A close-up of a chart

Description automatically generated

A black background with white text

Description automatically generated

**Deep Learning Overview**:

Deep learning powers various AI applications like self-driving cars, computer vision, and speech recognition. It's used for classification and unsupervised learning tasks. The course covers neural networks, advanced topics like CNNs and GANs, and reinforcement learning.

**Neural Network Basics:**

Neural networks are vital for AI applications. They process data in layers and use activation functions like the sigmoid. They relate to logistic regression, allowing non-linear patterns.

**Sigmoid Activation Function:**

The sigmoid activation function is crucial due to its differentiability. Its derivative is computed using the quotient rule, which involves exponential terms. The derivative relates to the function itself.

**Perceptron and MLP:**

A perceptron is a neural network's building block, like logistic regression. It transforms input through weights and the sigmoid function. Stacking layers creates more complex decision boundaries.

**Building an MLP:**

The course introduces building an MLP with Scikit-Learn, emphasizing hidden layers and activation functions. Future lessons will cover more advanced models with Keras.

**Navigating a Neural Network:**

Weights combine layers in a neural network. Inputs go through transformations, yielding activation values. These values progress from input to output, driven by weights and activations.

**Data Transformation in MLP:**

Data undergoes transformations through an MLP. Input values are combined with weights to create Z-values, transformed by activations to A-values. This process scales to handle datasets.

**Deep Learning Models:**

Deep learning includes neural networks, RNNs, CNNs, and unsupervised models like autoencoders and GANs. These models have diverse applications beyond traditional tasks.

**Gradient Descent:**

Gradient descent optimizes neural network parameters by minimizing a cost function. It starts with initialization and updates parameters iteratively. Learning rate controls the step size.

**Stochastic Gradient Descent:**

Stochastic gradient descent updates parameters using individual data points, introducing randomness.

**Mini-batch** gradient descent balances efficiency and stability by using subsets of data for updates.

A graph of a curve

Description automatically generated with medium confidence

A graph of a function

Description automatically generated

A graph of a function

Description automatically generated

A graph and diagram of a function

Description automatically generated

A graph of a function

Description automatically generated with medium confidence

A graph of a function

Description automatically generated with medium confidence

Feedforward computation and backpropagation are two fundamental concepts in neural network training:

1. **Feedforward Computation**: Feedforward computation refers to the process of passing input data through the neural network to obtain predictions or outputs. It involves propagating the input data forward through the network, layer by layer, applying weights and activation functions at each layer, until reaching the output layer. In this process, each layer computes its output based on the weighted sum of inputs from the previous layer and applies an activation function to produce the output.
2. **Backpropagation**: Backpropagation, short for "backward propagation of errors," is the primary algorithm used to train neural networks. It is based on the principle of gradient descent optimization. Backpropagation involves two main steps:

a. **Forward Pass**: During the feedforward computation, input data is propagated forward through the network to generate predictions.

b. **Backward Pass**: In the backward pass, the error or loss between the predicted output and the actual target is calculated. This error is then propagated backward through the network, layer by layer, to compute the gradient of the loss function with respect to the weights of the network. This is done using the chain rule of calculus. The gradients indicate how much each weight should be adjusted to minimize the error.

The computed gradients are then used to update the weights of the network in the direction that minimizes the loss, typically using optimization algorithms such as stochastic gradient descent (SGD) or its variants.

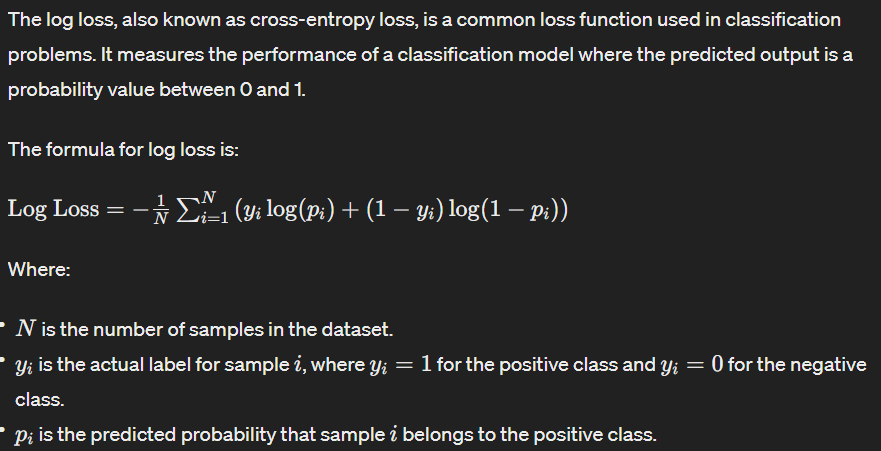
By iteratively performing feedforward computation and backpropagation, neural networks learn to adjust their weights to minimize the difference between predicted and actual outputs, thereby improving their performance on the given task. This process continues until convergence or a predefined stopping criterion is met.

Sigmoid Activation Function:

A computer screen with text and symbols

Description automatically generated

LogLoss:



A computer screen with white and pink text

Description automatically generated

A close-up of a function summary

Description automatically generated

**# Steps towards building creating a multi-layer perceptron**

1. Initialize the weights to random values between -1 and 1.
2. Perform the feed-forward computation.
3. Compute the loss function.
4. Calculate the gradients for all the weights via back-propagation.
5. Update the weight matrices (using a learning\_rate parameter).
6. Execute steps 2-5 for a fixed number of iterations.
7. Plot the accuracies and log loss and observe how they change over time

**# Sigmoid Activation:**

A close up of text

Description automatically generatedA graph with a line

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

### Optimizers in Deep Learning:

**What is an optimizer?**

**Optimizers** are algorithms or methods used to minimize an error function(*loss function*)or to maximize the efficiency of production. Optimizers are mathematical functions which are dependent on model’s learnable parameters i.e Weights & Biases. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

This post will walk you through the optimizers and some popular approaches.

**Types of optimizers**

Let’s learn about different types of optimizers and how they exactly work to minimize the loss function.

**Gradient Descent**

Gradient descent is an optimization algorithm based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum. Gradient Descentiteratively reduces a loss function by moving in the direction opposite to that of steepest ascent. It is dependent on the derivatives of the loss function for finding minima. uses the data of the entire training set to calculate the gradient of the cost function to the parameters which requires large amount of memory and slows down the process.

A graph of a line

Description automatically generated

Gradient Descent

A black background with a black square

Description automatically generated with medium confidence

**Advantages of Gradient Descent**

1. Easy to understand
2. Easy to implement

**Disadvantages of Gradient Descent**

1. Because this method calculates the gradient for the entire data set in one update, the calculation is very slow.
2. It requires large memory and it is computationally expensive.

**Learning Rate**

How big/small the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.

A graph of a function

Description automatically generated

Learning Rate

**Stochastic Gradient Descent**

It is a variant of Gradient Descent. It update the model parameters one by one. If the model has 10K dataset SGD will update the model parameters 10k times.

A diagram of a graph

Description automatically generated

Stochastic Gradient Descent

**Advantages of Stochastic Gradient Descent**

1. Frequent updates of model parameter
2. Requires less Memory.
3. Allows the use of large data sets as it has to update only one example at a time.

**Disadvantages of Stochastic Gradient Descent**

1. The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
2. High Variance.
3. Frequent updates are computationally expensive.

**Mini-Batch Gradient Descent**

It is a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. it can reduce the variance when the parameters are updated, and the convergence is more stable. It splits the data set in batches in between 50 to 256 examples, chosen at random.

A diagram of a spiral

Description automatically generated

Mini Batch Gradient Descent

**Advantages of Mini Batch Gradient Descent:**

1. It leads to more stable convergence.
2. more efficient gradient calculations.
3. Requires less amount of memory.

**Disadvantages of Mini Batch Gradient Descent**

1. Mini-batch gradient descent does not guarantee good convergence,
2. If the learning rate is too small, the convergence rate will be slow. If it is too large, the loss function will oscillate or even deviate at the minimum value.

**SGD with Momentum**

**SGD with Momentum** is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent. Momentum simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction. In this way, you can increase the stability to a certain extent, so that you can learn faster, and also have the ability to get rid of local optimization.

A diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of a diagram of

Description automatically generated

SGD with Momentum

A black background with a black square

Description automatically generated with medium confidence

Momentum Formula

**Advantages of SGD with momentum**

1. Momentum helps to reduce the noise.
2. Exponential Weighted Average is used to smoothen the curve.

**Disadvantage of SGD with momentum**

1. Extra hyperparameter is added.

**AdaGrad(Adaptive Gradient Descent)**

In all the algorithms that we discussed previously the learning rate remains constant. The intuition behind AdaGrad is can we use different Learning Rates for each and every neuron for each and every hidden layer based on different iterations.

A black background with a black square

Description automatically generated with medium confidence

**Advantages of AdaGrad**

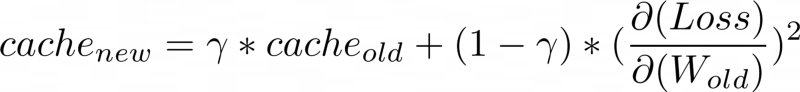
1. Learning Rate changes adaptively with iterations.
2. It is able to train sparse data as well.

**Disadvantage of AdaGrad**

1. If the neural network is deep the learning rate becomes very small number which will cause dead neuron problem.

**RMS-Prop (Root Mean Square Propagation)**

RMS-Prop is a special version of Adagrad in which the learning rate is an exponential average of the gradients instead of the cumulative sum of squared gradients. RMS-Prop basically combines momentum with AdaGrad.



**Advantages of RMS-Prop**

1. In RMS-Prop learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

**Disadvantages of RMS-Prop**

1. Slow Learning

**AdaDelta**

Adadelta is an extension of Adagrad and it also tries to reduce Adagrad’s aggressive, monotonically reducing the learning rate and remove decaying learning rate problem. In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

**Advantages of Adadelta**

1. The main advantage of AdaDelta is that we do not need to set a default learning rate.

**Disadvantages of Adadelta**

1. Computationally expensive

**Adam(Adaptive Moment Estimation)**

Adam optimizer is one of the most popular and famous gradient descent optimization algorithms. It is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients , similar to momentum and also the decaying average of the past squared gradients , similar to RMS-Prop and Adadelta. Thus, it combines the advantages of both the methods.

A black and white math equations

Description automatically generated with medium confidence

**Advantages of Adam**

1. Easy to implement
2. Computationally efficient.
3. Little memory requirements.

**Comparison**

A colorful lines with black dots

Description automatically generated with medium confidence

Optimizers Comparison

A graph of a function

Description automatically generated with medium confidence

**Optimization on saddle point**

**How to choose optimizers?**

* If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
* RMSprop, Adadelta, Adam have similar effects in many cases.
* Adam just added bias-correction and momentum on the basis of RMSprop,
* As the gradient becomes sparse, Adam will perform better than RMSprop

A computer screen shot of a chart

Description automatically generated

## CNN (Convolution neural Networks):

**Preprocessing and Data Preparation:**

* Preprocessing is vital for making data work well with neural networks.
* In multiclass classification, the last network layer changes to match the number of classes.
* The SoftMax function helps determine the class probabilities.
* The categorical cross-entropy loss function gauges how well the model's predictions match reality.

**Introduction to Convolutional Neural Networks (CNNs):**

* CNNs are awesome for recognizing images and patterns.
* They mimic how our eyes detect features and shapes in images.
* They're great at capturing edges, patterns, and shapes.
* CNNs can work not only on images but also on other kinds of data.

**Using Kernels in CNNs:**

* Kernels are like tools that CNNs use to understand images.
* They slide over images, figuring out edges and patterns.
* Kernels help CNNs find features that are important for recognizing things.

**Color Images and Filters in CNNs:**

* For color images, we use filters to understand different color channels.
* Filters help CNNs see patterns in red, green, and blue parts of images.
* They help CNNs recognize colors and features in images.

**Grid Size, Padding, and Stride in CNNs:**

* Grid size is about how much a CNN looks at in one go.
* Padding is like adding a border to images so CNNs can see the edges better.
* Stride is how much a CNN moves while looking at an image.

**Channels, Depth, Filters, and Pooling in CNNs:**

* Images have different layers representing colors and features.
* Filters help CNNs understand images better by looking at small parts.
* Pooling simplifies images by focusing on the important parts.

## **Transfer Learning**

### # What is transfer learning?

Most popular models are difficult to train from scratch as they require huge datasets (like ImageNet), a large number of training iterations and very heavy computing machinery. The basic features (edges, shapes) learned by early layers in a network are generalizable. While the later layers in an already trained network tend to capture features that are more particular to a specific image classification task. Transfer learning uses the idea that if we keep the early layers of a pre-trained network, and re-train the later layers on a specific dataset, we might be able to leverage some state of that network on a related task.

### # A typical transfer learning workflow in Keras

1. Initialize base model, and load pre-trained weights (e.g. ImageNet).
2. "Freeze" layers in the base model by setting `training = False`.
3. Define a new model that goes on top of the output of the base model's layers.
4. Train resulting model on your data set.

### # Fine tuning in transfer learning

Fine-tuning is an optional step in transfer learning, it usually ends up improving the performance of the model. It is easy to overfit the model in this step as we are re-training the entire model. So we use regularization (dropout layers), a lower learning rate, a small number of epochs (training iterations), and early stopping to know when the model has stopped improving and to prevent overfitting.

## **# Modern CNN Architectures**

### # Inception - V3

Instead of focusing on increasing the depth of the network, InceptionNet focuses on increasing the width and depth of the model simultaneously to attain better accuracy, while keeping the computing resources constant. It focuses on **parallel processing** and extraction of various feature maps concurrently using **Inception modules**, which are collections of convolutions with different filter sizes and pooling operations. The following is an illustration of the inception module in inception-v1 architecture:

A diagram of a diagram

Description automatically generated

### # ResNet - 50

**ResNet** features special skip connections which add the output from an earlier layer directly to a later layer and a heavy use of batch normalization. It allows us to design deep CNNs without compromising the model’s convergence and accuracy. The basic building blocks for ResNets is the convolution and identity blocks. Essentially, ResNet uses the network layers to fit a residual mapping �(�)+�,*F*(*x*)+*x*, instead of trying to learn the desired underlying mapping $H(x)$ directly with stacked layers.

A diagram of a diagram

Description automatically generated

## **# Regularization Techniques**

* L2 (Ridge) regularization
* L1 (Lasso) regularization
* Dropout
* Batch Normalization
* Data shuffling

## Regularization

A common issue with training deep neural networks or any machine learning models is **overfitting**, which refers to a situation where the model performs too well on the training data but fails to generalize to the test data.

To combat overfitting or high variance, we introduce 4 regularization techniques in this reading for you to choose from the next time you train your neural network. They are:

* Batch Normalization
* Dropout
* L1 regularization (Lasso)
* L2 regularization (Ridge)

### 1. Batch Normalization

Training deep neural networks is complicated, because the distribution of each layer’s inputs changes as the weights of the previous layers get updated during back propagation. This can result in the learning algorithm always pursuing a moving target. In the [original paper](https://arxiv.org/abs/1502.03167) of Batch Normalization, the change in the distribution of inputs to lsayers in the network is referred to as **“internal covariate shift”**. This slows down the training by requiring lower learning rates and careful parameter initialization, making it hard for the model to generalize well. Sometimes, if the training images are statistically too different from the testing images, it is considered a covariate shift.

#### What is Batch Normalization?

Batch normalization is a technique for training very deep neural networks that normalizes the inputs to a layer for every mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

**Batch**, or **mini-batch**, is a collection of samples that will be passed through the network at one time for the weights update. **Normalization** is the process of transforming the data to have a mean 0 and standard deviation 1 (thus follows the standard normal distribution).

In a neural network, batch normalization is achieved through a normalization layer in between the fully connected Dense layers, which fixes the means and variances of each layer’s inputs. This way, there is not much change in the distrbution of each layer input, and the layers in the network can learn from back-propagation simultaneously with out having to wait for the previous layers to learn. This accelerates the training process.

A diagram of a network

Description automatically generated

Image credits to [Ilango Rajagopal](https://medium.com/@ilango100/batch-normalization-speed-up-neural-network-training-245e39a62f85)

#### How does Batch Normalization work?

Denote by � a mini-batch of size � of the training set, for a layer with **d**-dimensional input �=(�(1),...,�(�)), each dimension of its input is then normalized as:�^��=���−������2+�where �∈[1,�]and �∈[1,�]; ��� and ���2are the **per dimension mean and variance** of � such that���=1�∑�=1����,���2=1�∑�=1�(���−���)2� is added in the denominator for numerical stability and is an arbitrarily small constant. The resulting normalized activation �^� have zero mean and unit variance, if � is not taken into account. To restore the representation power of the network, a transformation step then follows as

���=���^��+��where the parameters �� and �� are subsequently learned in the optimization process. You can refer to [wikipedia](https://en.wikipedia.org/wiki/Batch_normalization) for more mathematical details of batch normalization.

What are the benefits of Batch Normalization?

1. It tackles the internal covariate shift issue by always normalizing the input signals, thus accelerating the training of deep neural nets and increasing the generalization power of the networks.
2. It makes the optimization landscape smoother, reduing the magnitude of variations in the gradient and the loss, making the training faster and easier.
3. It acts as a regularizer by introducing random noises, as each minibatch at each layer is normalized using its own mean and standard deviation during training.

#### Using Batch Normalization in Keras

In Keras, you can add a Batch Norm layer in between the Dense layers by calling BatchNormalization() from **keras.layers**. Here is a code example:

from tensorflow.keras.layers import Dense, BatchNormalization

model = Sequential([

Dense(64, input\_shape=(4,), activation="relu"),

BatchNormalization(),

Dense(128, activation='relu'),

BatchNormalization(),

Dense(64, activation='relu')

])

### 2. Dropout

Dropout is a regularization technique that approximates training a large number of different neural nerworks in parallel. During training, using dropout will keep a neuron active with some probability � (a hyperparameter), or set it to zero otherwise.

A diagram of a neural network

Description automatically generated

Picture from the [Dropout paper](http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf)

Dropout can be interpreted as sampling a neural network within the full neural network, and only updating the weights of the sampled network. By doing so, each iteration is using a different model architecture and has a different “view” of the configured layers. This adds noise to the training process, and perhaps breaks up situations where network layers co-adapt to correct mistakes from prior, in turn making the network more robust.

Note that dropout is not used during prediction.

#### Using Dropout in Keras

In Keras, Dropout can be applied using the Dropout class from **keras.layers**. A dropout rate can be specified when creating the dropout layer, which is the percentage of neurons that will be turned off during one update.

from tensorflow.keras.layers import Dropout

model.add(Dropout(rate=0.2))

### 3. L2 Regularization

L2 regularization is perhaps the most common form of regularization and is also known as **Ridge regression**. It penalizes the squared magnitude of the weights � by adding the term 12��2 to the objective function that the algorithm is trying to optimize. � is the regularization strength, also called the **shrinkage parameter** which can be tuned during training.

The regularization term is defined as the Euclidean norm of the weight matrices, which sums over all the squared weights. It is multiplied by 12 so that when the gradient is calculated we get ��.

As the shrinkage parameter approches infinity, the weights are driven down to near zero but **will not be exactly zero**.

#### Using L2 Regularization in Keras

In Keras, regularization penalties are applied on a per-layer basis, that is, you specify the name of the regularizer API as you add a layer to the network. The tf.keras.regularizers module has a built-in L2 class that you can call using the following:

*# regularization strength is specified through the L1 argument*

tf.keras.regularizers.L2(l2=0.01)

This object can be passed as an argument to an added layer:

model.add(Dense(32,

activation="relu",

kernel\_regularizer=tf.keras.regularizers.L2(l2=0.01)))

### 4. L1 Regularization

L1 regularization is another common form of regularization. It penalizes large weights � by adding the term �|�| to the objective function, where � is the regularization strength.

What’s differs L1 from L2 is that, when � approches infinity, L1 regularization can shrink the weights of the less important features to zero, which would be very useful if you also want automatic feature selection during training.

The following figure illustrates how L1 and L2 work differently:

A diagram of a galaxy

Description automatically generated with medium confidence

Source: An Introduction to Statistical Learning

In the figure above, the red ellipses represent the contours of the loss function that needs to be optimized during training, and the green areas on the left and right represent the **feasible regions** of the L1 and L2 constaints respectively.

In the case of L1 (left), the ellipse would be able to intersect with the feasible region on an axis, which is when the weights become zero. However, for L2 (right), due to the round shape of the feasible region, i.e., the L2 norm, the ellipse wouldn’t intersect with the green circle on an axis, which is why the weights regularized by L2 can be close to zero but not equal to zero.

We can also combine the L1 regularization and the L2 regularization to obtain the **Elastic Net regularization**: 12�1�2+�2|�|. �1 and �2 can be tuned as hyperparameters during training.

#### Using L1 Regularization in Keras

Similar to how we used L2, tf.keras.regularizers module also has a built-in L1 class that you can use to regularize the weights of a layer:

model.add(Dense(32,

activation="relu",

kernel\_regularizer=tf.keras.regularizers.L1(l1=0.01)))

*# If you don't need to specify a value for the regularization strength,*

*# you can also do this instead:*

model.add(Dense(32,

activation="relu",

kernel\_regularizer="l1"))

### **Recurrent Neural Networks (RNNs)**

Recurrent Neural Networks are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are mostly used in applications of natural language processing and speech recognition.

One of the main motivations for RNNs is to derive insights from text and do better than “bag of words” implementations. Ideally, each word is processed or understood in the appropriate context.

Words should be handled differently depending on “context”. Also, each word should update the context.

Under the notion of recurrence, words are input one by one. This way, we can handle variable lengths of text. This means that the response to a word depends on the words that preceded it.

These are the two main outputs of an RNN:

* Prediction: What would be the prediction if the sequence ended with that word.
* State: Summary of everything that happened in the past.

**Mathematical Details**

Mathematically, there are cores and subsequent dense layers

current state = function1(old state, current input).

current output = function2(current state).

We learn function1 and function2 by training our network!

r = dimension of input vector

s = dimension of hidden state

t = dimension of output vector (after dense layer)

  U is a s × r matrix

W is a s × s matrix

V is a t × s matrix

In which the weight matrices U, V, W are the same across all positions

**Practical Details**

Often, we train on just the ”final” output and ignore intermediate outputs.

Slight variation called Backpropagation Through Time (BPTT) is used to train RNNs.

Sensitive to length of sequence (due to “vanishing/exploding gradient” problem).

In practice, we still set a maximum length to our sequences. If the input is shorter than maximum, we “pad” it. If the input is longer than maximum, we truncate it.

**RNN Applications**

RNNs often focus on text applications, but are commonly used for other sequential data:

* Forecasting: Customer Sales, Loss Rates, Network Traffic.
* Speech Recognition: Call Center Automation, Voice Applications.
* Manufacturing Sensor Data
* Genome Sequences

**Long-Short Term Memory RNNs (LSTM)**

LSTMs are a special kind of RNN (invented in 1997). LSTM has as motivation solve one of the main weaknesses of RNNs, which is that its transitional nature, makes it hard to keep information from distant past in current memory without reinforcement.

LSTM have a more complex mechanism for updating the state.

Standard RNNs have poor memory because the transition Matrix necessarily weakens signal.

This is the problem addressed by Long-Short Term Memory RNNs (LSTM).

To solve it, you need a structure that can leave some dimensions unchanged over many steps.

* By default, LSTMs remember the information from the last step.
* Items are overwritten as an active choice.

The idea for updating states that RNNs use is old, but the available computing power to do it sequence to sequence mapping, explicit memory unit, and text generation tasks is relatively new.

Augment RNNs with a few additional Gate Units:

* Gate Units control how long/if events will stay in memory.
* Input Gate: If its value is such, it causes items to be stored in memory.
* Forget Gate: If its value is such, it causes items to be removed from memory.
* Output Gate: If its value is such, it causes the hidden unit to feed forward (output) in the network.

**Gated Recurrent Units (GRUs)**

GRUs are a gating mechanism for RNNs that is an alternative to LSTM. It is based on the principle of Removed Cell State:

* Past information is now used to transfer past information.
* Think of as a “simpler” and faster version of LSTM.

These are the gates of GRU:

Reset gate: helps decide how much past information to forget.

Update gate: helps decide what information to throw away and what new information to keep.

**LSTM vs GRU**

LSTMs are a bit more complex and may therefore be able to find more complicated patterns.

Conversely, GRUs are a bit simpler and therefore are quicker to train.

GRUs will generally perform about as well as LSTMs with shorter training time, especially for smaller datasets.

In Keras it is easy to switch from one to the other by specifying a layer type. It is relatively quickly to change one for the other.

**Sequence-to-Sequence Models (Seq2Seq)**

Thinking back to any type of RNN interprets text, the model will have a new hidden state at each step of the sequence containing information about all past words.

Seq2Seq improve keeping necessary information in the hidden state from one sequence to the next.

This way, at the end of a sentence, the hidden state will have all information relating to past words.

The size of the vector from the hidden state is the same no matter the size of the sentence.

In a nutshell, there is an encoder, a hidden state, and a decoder.

**Beam Search**

Beam search is an attempt to solve greedy inference.

* Greedy Inference, which means that a model producing one word at a time implies that if it produces one wrong word, it might output a wrong entire sequence of words.
* Beam search tries to produce multiple different hypotheses to produce words until <EOS> and then see which full sentence is most likely.

These are examples of common enterprise applications of LSTM models:

* Forecasting: (LSTM among most common Deep Learning models used in forecasting).
* Speech Recognition
* Machine Translation
* Image Captioning
* Question Answering
* Anomaly Detection
* Robotic Control

## Gated RNN

The two types of gated RNNs we will be studying in this lab are Long Short Term Memories (LSTM) and Gated Recurrent Units (GRU). GRU is simpler than LSTM. It's much faster and optimizes quicker.

### LSTM

The key idea of LSTMs is to have two state representations: the hidden state 𝐡ℎ and the cell state 𝐂� (instead of 𝐬�).

An LSTM cell has a complex internal structure that makes it able to:

* learn to recognize an important input,
* store it in the long-term state,
* preserve it for as long as it is needed,
* extract it whenever it is needed.

A diagram of a diagram

Description automatically generated

Source: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called gates, which are a way to optionally let information through. There are three gates in an LSTM; forget gate, input gate, and output gate. They are composed out of a sigmoid neural net layer. Sigmoid-based layers output values near either 0 (gate closed) or 1 (gate open).

### GRU

GRU is a simplification of the LSTM cell that performs similarly well while being faster to train. It has a single update gate controller that manages both the forget and input gates. Whenever one is open, the other is closed. There is no output gate. The reset gate controls which part of the previous state is shown to the main layer. The resulting model is simpler than standard LSTM models. GRUs have fewer parameters and thus may train a bit faster or need less data to generalize.

A diagram of a complex equation

Description automatically generated

**Seq to seq models** - Encoders Decoders using RNN or LSTM or GRU

<https://www.baeldung.com/cs/rnns-transformers-nlp>

Attention is also used when we need the words passed to encoder should be meaningful

Here BiFirectional RNN is used.

A person pointing at a whiteboard

Description automatically generated

A diagram of a machine

Description automatically generated

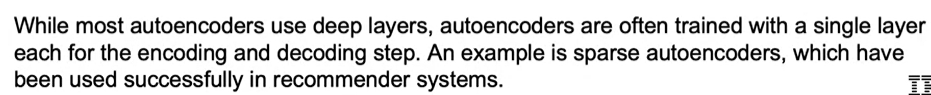
<https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html>

A white background with black text

Description automatically generated

A close-up of a business application

Description automatically generated



**Autoencoders**

Autoencoders are a neural network architecture that forces the learning of a lower dimensional representation of data, commonly images.

Autoencoders are a type of unsupervised deep learning model that use hidden layers to decompose and then recreate their input. They have several applications:

* Dimensionality reduction
* Preprocessing for classification
* Identifying ‘essential’ elements of the input data, and filtering out noise

One of the main motivations is find whether two pictures are similar.

**Autoencoders and PCA**

Autoencoders can be used in cases that are suited for Principal Component Analysis (PCA).

Autoencoders also help to deal with some of these PCA limitations: PCA has learned features that are linear combinations of original features.

Autoencoders can detect complex, nonlinear relationship between original features and best lower dimensional representation.

**Autoencoding process**

The process for autoencoding can be summarized as:

1. Feed image through encoder network
2. Generate the lower dimension embedding
3. Feed embedding through decoder network
4. Generate reconstructed version of the original data
5. Compare the result of the generated vs the original image

Result: A network will learn the lower dimensional space that represents the original data

**Autoencoder applications**

Autoencoders have a wide variety of enterprise applications:

* Dimensionality reduction as preprocessing for classification
* Information retrieval
* Anomaly detection
* Machine translation
* Image-related applications (generation, denoising, processing and compression)
* Drug discovery
* Popularity prediction for social media posts
* Sound and music synthesis
* Recommender systems

A diagram of a mathematical equation

Description automatically generated

A line of yellow and white objects with text

Description automatically generated with medium confidence

A close up of text

Description automatically generated

A screenshot of a computer

Description automatically generated

A diagram of a rectangular object

Description automatically generated

## **Variational Autoencoders**

Variational autoencoders also generate a latent representation and then use this representation to generate new samples (i.e. images).

These are some important features of variational autoencoders:

* Data are assumed to be represented by a set of normally-distributed latent factors.
* The encoder generates parameters of these distributions, namely µ and σ.
* Images can be generated by sampling from these distributions.

## **VAE goals**

The main goal of VAEs: generate images using the decoder

The secondary goal is to have similar images be close together in latent space

## **Loss Function of Variational Autoencoders**

The VAE reconstruct the original images from the space of vectors drawn from a standard normal distribution.

The two components of the loss function are:

* A penalty for not reconstructing the image correctly.
* A penalty for generating vectors of parameters µ and σ that are different than 0 and 1, respectively: the parameters of the standard normal distribution.

## **Generative Adversarial Networks (GANs)**

The invention of GANs was connected to neural networks’ vulnerability to adversarial examples. Researchers were going to run a speech synthesis contest, to see which neural network could generate the most realistic-sounding speech.

A neural network - the “discriminator” - would judge whether the speech was real or not.

In the end, they decided not to run the contest, because they realized people would generate speech to fool this particular network, rather than actually generating realistic speech.

**These are the step to train GANs:**

* Randomly initialize weights of generator and discriminator networks
* Randomly initialize noise vector and generate image using generator
* Predict probability generated image is real using discriminator
* Compute losses both assuming the image was fake and assuming it was real
* Train the discriminator to output whether the image is fake
* Compute the penalty for the discriminator probability, without using it to train the discriminator
* Train the generator to generate images that the discriminator thinks are real
* Use the discriminator to calculate the probability that a real image is real
* Use L to train the discriminator to output 1 when it sees real images

## **Reinforcement Learning**

In Reinforcement Learning, Agents interact with an Environment

They choose from a set of available Actions

The actions impact the Environment, which impacts agents via Rewards

Rewards are generally unknown and must be estimated by the agent

The process repeats dynamically, so agents learn how to estimate rewards over time

Advances in deep learning have led to many recent RL developments:

* In 2013, researchers from DeepMind developed a system to play Atari games
* In 2017, the AlphaGo system defeated the world champion in Go

In general, RL algorithms have been limited due to significant data and computational requirements.

As a result, many well-known use cases involve learning to play games. More recently, progress has been made in areas with more direct business applications.

## **Reinforcement Learning Architecture**

The main components of reinforcement learning are: Policy, Agents, Actions, State, and Reward.

* Solutions represents a Policy by which Agents choose Actions in response to the State
* Agents typically maximize expected rewards over time
* In Python, the most common library for RL is Open AI GYM

This differs from typical Machine Learning Problems:

* Unlike labels, rewards are not known and are often highly uncertain
* As actions impact the environment, the state changes, which changes the problem
* Agents face a tradeoff between rewards in different periods

Examples of everyday applications of Reinforcement Learning include recommendation engines, marketing, and automated bidding.