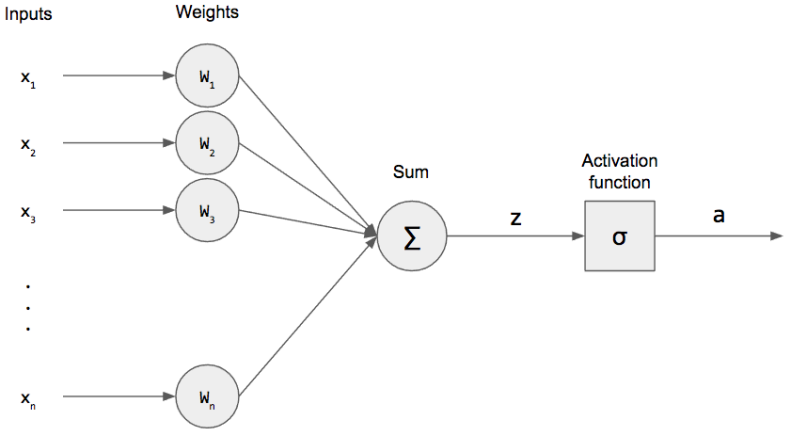
ACV-Question Bank

Q1) Explain AND and NAND gates using peceptron..



AN

X1	X2	output
0	0	0
0	1	0
1	0	0
1	1	1

X1	X2	output
0	0	1
0	1	1
1	0	1
1	1	0

First, we need to know that the Perceptron algorithm states that:
Prediction (y`) = 1 if Wx+b >= 0 and 0 if Wx+b<0

The steps in this method are very similar to how Neural Networks learn, which is as follows;

- Initialize weight values and bias
- Forward Propagate
- Check the error
- Backpropagate and Adjust weights and bias
- Repeat for all training examples

Row 1

From w1x1+w2x2+b, initializing w1, w2, as 1 and b as -1, we get: x1(1)+x2(1)-1

Passing the first row of the AND logic table (x1=0, x2=0), we get: 0+0–1 = -1

From the Perceptron rule, if Wx+b<0, then y`=0. Therefore, this row is correct, and no need for Backpropagation.

Row 2

Passing (x1=0 and x2=1), we get: 0+1–1 = 0

From the Perceptron rule, if Wx+b >= 0, then y`=1. This row is incorrect, as the output is 0 for the AND gate. So, we want values that will make the combination of x1=0 and x2=1 to give y` a value of 0. If we change b to -1.5, we have: 0+1–1.5 = -0.5

From the Perceptron rule, this works (for both row 1, row 2 and 3).

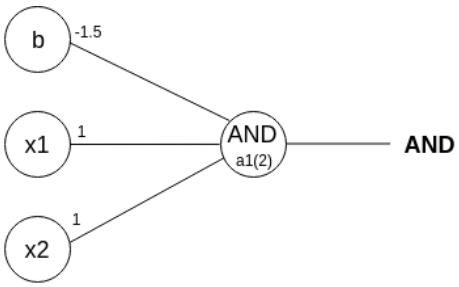
Row 4

Passing (x1=1 and x2=1), we get: 1+1–1.5 = 0.5

Again, from the perceptron rule, this is still valid.

Therefore, we can conclude that the model to achieve an AND gate, using the Perceptron algorithm is: **x1+x2–1.5**

ACV-Question Bank



NAND Gate

Row 1

From w1x1+w2x2+b, initializing w1 and w2 as 1, and b as -1, we get: x1(1)+x2(1)-1

Passing the first row of the NAND logic table (x1=0, x2=0), we get: 0+0-1 = -1

From the Perceptron rule, if Wx+b<0, then y`=0. This row is incorrect, as the output is 1 for the NAND gate. So, we want values that will make input x1=0 and x2 = 0 to give y` a value of 1. If we change b to 1, we have: 0+0+1 = 1. From the Perceptron rule, this works.

Row 2

Passing (x1=0, x2=1), we get: 0+1+1 = 2

From the Perceptron rule, if Wx+b >= 0, then y`=1. This row is also correct (for both row 2 and row 3).

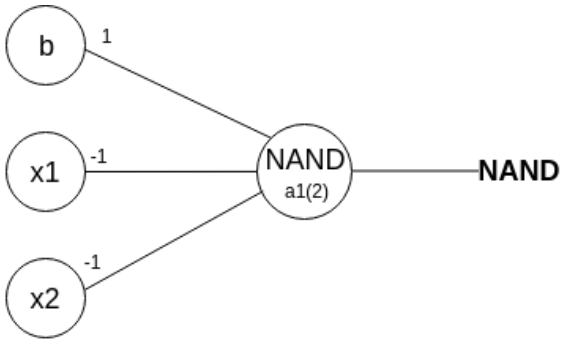
Row 4

Passing (x1=1, x2=1), we get: 1+1+1 = 3

This is not the expected output, as the output is 0 for a NAND combination of x1=1 and x2=1.

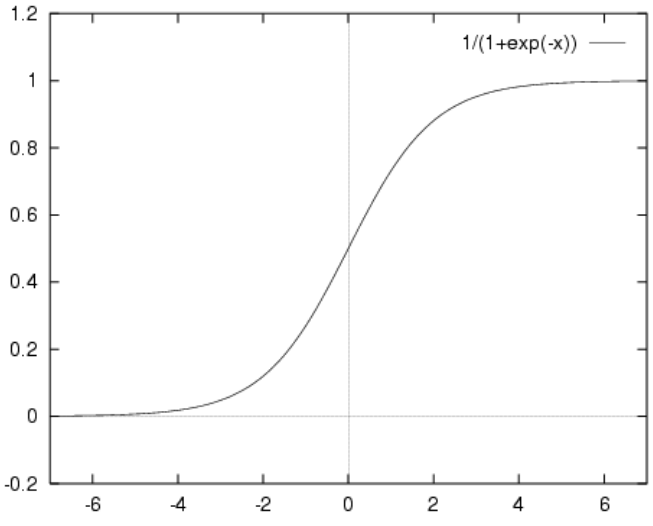
Changing values of w1 and w2 to -1, we get: -1-1+1 = -1

Therefore, we can conclude that the model to achieve a NAND gate, using the Perceptron algorithm is: -**x1-x2+1**



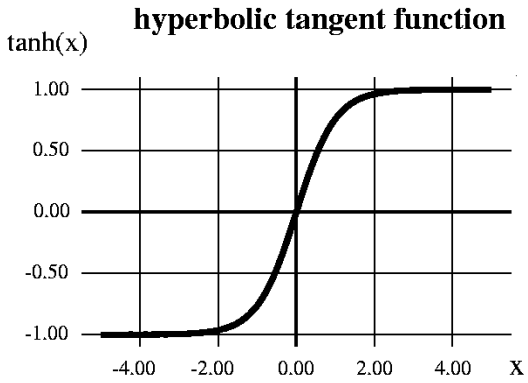
Q2) Draw graph & eq. for atleast 5 popular activation functions.

Sigmoid Activation function: It is a activation function of form f(x) = 1 / 1 + exp(-x) . Its Range is between 0 and 1. It is a S — shaped curve. It is easy to understand and apply.

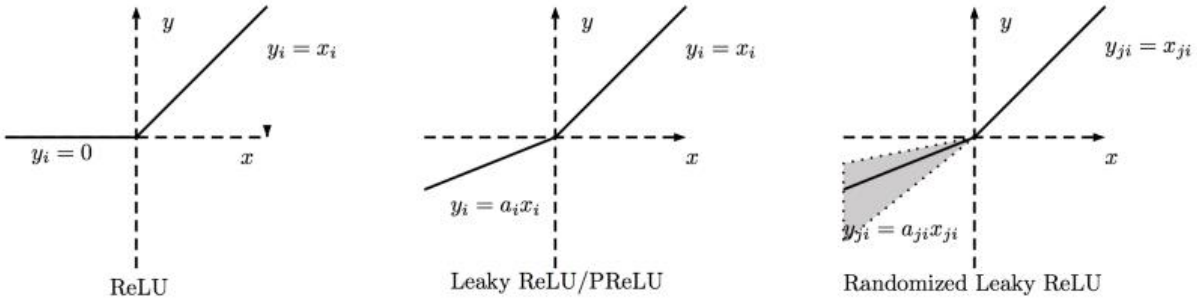


ACV-Question Bank

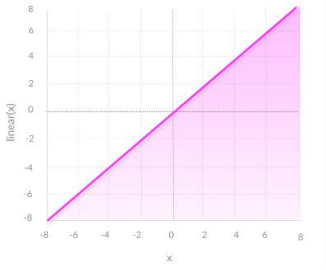
Hyperbolic Tangent function- Tanh : It's mathematical formula is f(x) = 1 — exp(-2x) / 1 + exp(-2x). Now it's output is zero centred because its range in between -1 to 1 i.e -1 < output < 1 . Hence optimization is easier in this method hence in practice it is always preferred over Sigmoid function.



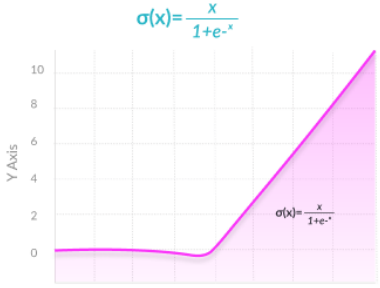
ReLu- Rectified Linear units : It has become very popular in the past couple of years. It was recently proved that it had 6 times improvement in convergence from Tanh function. It's just R(x) = max(0,x) i.e if x < 0 , R(x) = 0 and if x >= 0 , R(x) = x. Hence as seeing the mathematical form of this function we can see that it is very simple and efficient.



Linear activation: A linear activation function takes the form: **A = cx**



Swish activation: Swish is a new, self-gated activation function discovered by researchers at Google. According to their paper, it performs better than ReLU with a similar level of computational efficiency.



ACV-Question Bank

Q3) What are the general steps in gradient descent optimization?

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks.

Starting at the top of the mountain, we take our first step downhill in the direction specified by the negative gradient. Next, we recalculate the negative gradient (passing in the coordinates of our new point) and take another step in the direction it specifies. We continue this process iteratively until we get to the bottom of our graph, or to a point where we can no longer move downhill–a local minimum.

Learning rate: The size of these steps is called the learning rate. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing.

Cost function: A Loss Functions tells us "how good" our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to update our parameters to make the model more accurate.

Given the cost function: **$f(m,b)=1N\sum_{i=1}^{n}(y_i-(mx_i+b))^2$**

The gradient can be calculated as: **$f'(m,b)=\begin{bmatrix}\frac{df}{dm}\\ \frac{df}{db}\end{bmatrix}=[1N\sum-2x_i(y_i-(mx_i+b))\ 1N\sum-2(y_i-(mx_i+b))]$**

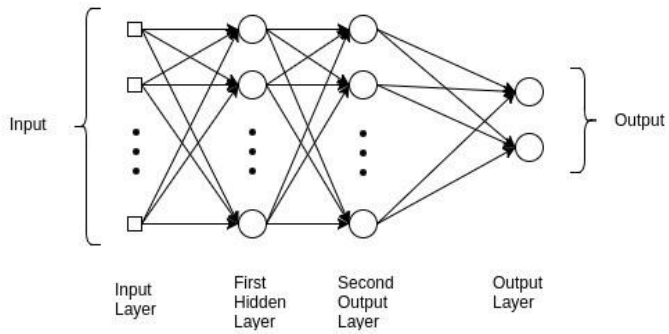
To solve for the gradient, we iterate through our data points using our new m and b values and compute the partial derivatives. This new gradient tells us the slope of our cost function at our current position (current parameter values) and the direction we should move to update our parameters. The size of our update is controlled by the learning rate.

Optimization Techniques

- **Momentum method:** This method is used to accelerate the gradient descent algorithm by taking into consideration the exponentially weighted average of the gradients. Using averages makes the algorithm converge towards the minima in a faster way, as the gradients towards the uncommon directions are cancelled out.
- **RMSprop:** RMSprop was proposed by University of Toronto's Geoffrey Hinton. The intuition is to apply an exponentially weighted average method to the second moment of the gradients (dW2).
- **Adam Optimization:** Adam optimization algorithm incorporates the momentum method and RMSprop, along with bias correction.

Q4) What is a Multi-Layer-Perceptron?

In the Perceptron Algorithm, we just multiply with weights and add Bias, but we do this in one layer only. We update the weight when we find an error in classification or miss-classified. Weight update equation is this = **weight = weight + learning_rate * (expected - predicted) * x**



Now comes to Multilayer Perceptron(MLP) or Feed Forward Neural Network(FFNN). In the Multilayer perceptron, there can more than linear layer or neurons. If we take the simple example the three-layer first will be the input layer and last will be output layer and middle layer will be hidden layer. We feed the input data to the input layer and take the output from the output layer. We can increase the number of the hidden layer as much as we want, to make the model more complex.

Feed Forward Network, is the most typical neural network model. Its goal is to approximate some function f (). Given, for example, a classifier $y = f*(x)$ that maps an input x to an output class y, the MLP find the best approximation to that classifier by defining a mapping, $y = f(x; \theta)$ and learning the best parameters, θ, for it. The MLP networks are composed of many functions that are, for instance, chained together. The layers of an MLP consists of several fully connected layers because each unit in a layer is connected to all the units in the previous layer. In a fully connected layer, the parameters of each unit are independent of the rest of the units in the layer, that means each unit possess a unique set of weights.

In a supervised classification system, each input vector is associated with a label, or ground truth, defining its class or class label is given with the data. The output of the network gives a class score, or prediction, for each input. To measure the performance of the classifier, the loss function is defined. The loss will be high if the predicted class does not correspond to the true class, it will be low otherwise. Sometimes the problem of overfitting and underfitting occurs at the time of training the model. In this case, Our model performs very well on training data but not on testing data. In order to train the network, an optimization procedure is required for this we need loss function and an optimizer. This procedure will find the values for the set of weights, W that minimizes the loss function.

A popular strategy is to initialize the weights to random values and refine them iteratively to get a lower loss. This refinement is achieved by moving on the direction defined by the gradient of the loss function. And it is important to set a learning rate defining the amount in which the algorithm is moving in every iteration.

Training the Model- There are basically three steps in the training of the model.

- **Forward pass:** In this step of training the model, we just pass the input to model and multiply with weights and add bias at every layer and find the calculated output of the model.
- **Calculate error or loss:** When we pass the data instance(or one example) we will get some output from the model that is called Predicted output(pred_out) and we have the label with the data that is real output or expected output(Expect_out). Based upon these both we calculate the loss that we have to backpropagate(using Backpropagation algorithm). There is various Loss Function that we use based on our output and requirement.
- **Backward pass:** After calculating the loss, we backpropagate the loss and updates the weights of the model by using gradient. This is the main step in the training of the model. In this step, weights will adjust according to the gradient flow in that direction.

Q5) Why do we normalize inputs for DL ?

Normalization is needed because it removes geometrical biases towards some of the dimensions of the data vectors. In this way every bit of data gets treated in a "fair" manner. Another way of posing this is to realize that all learning algorithms depend on numerical properties so one should try to avoid small numbers, large numbers, and large differences.

Normalization is important in ANNs because real data obtained from experiments and analysis most times are distant from each other. The effect is great because the common activation functions such as sigmoid, hyperbolic tangent and gaussian produce result that ranges between [0,1] or [-1,1]. It is

important to normalise the values to be in that range. The common normalization approach includes Statistical normalization (using mean and standard deviation) and Min-Max Normalization.

Q6) Explain any 2 weigh initialization techniques?

The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.

Zero initialization : In general practice biases are initialized with 0 and weights are initialized with random numbers, what if weights are initialized with 0 ? If all the weights are initialized with 0 , the derivative with respect to loss function is same for every w in W[l], thus all weights have same value in subsequent iterations. This makes hidden units symmetric and continues for all the n iterations i.e. setting weights to 0 does not make it better than a linear model. An important thing to keep in mind is that biases have no effect what so ever when initialized with 0.

Random initialization: Assigning random values to weights is better than just 0 assignment. But there is one thing to keep in my mind is that what happens if weights are initialized high values or very low values and what is a reasonable initialization of weight values.

a) If weights are initialized with very high values the term np.dot(W,X)+b becomes significantly higher and if an activation function like sigmoid() is applied, the function maps its value near to 1 where slope of gradient changes slowly and learning takes a lot of time.

b) If weights are initialized with low values it gets mapped to 0, where the case is same as above.

One method is HE Initialization $\sqrt{\frac{2}{size^{[l-1]}}}$

$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(2/size\_l\text{-}1)$$

Xavier initialization: It is same as He initialization but it is used for tanh() activation function, in this method 2 is replaced with 1. $\sqrt{\frac{1}{size^{[l-1]}}}$

$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(1/size\_l\text{-}1)$$

Q7) What is dropout ?

Dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By “ignoring”, I mean these units are not considered during a particular forward or backward pass.

More technically, at each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data.

Q8) What is Batch-Normalization ?

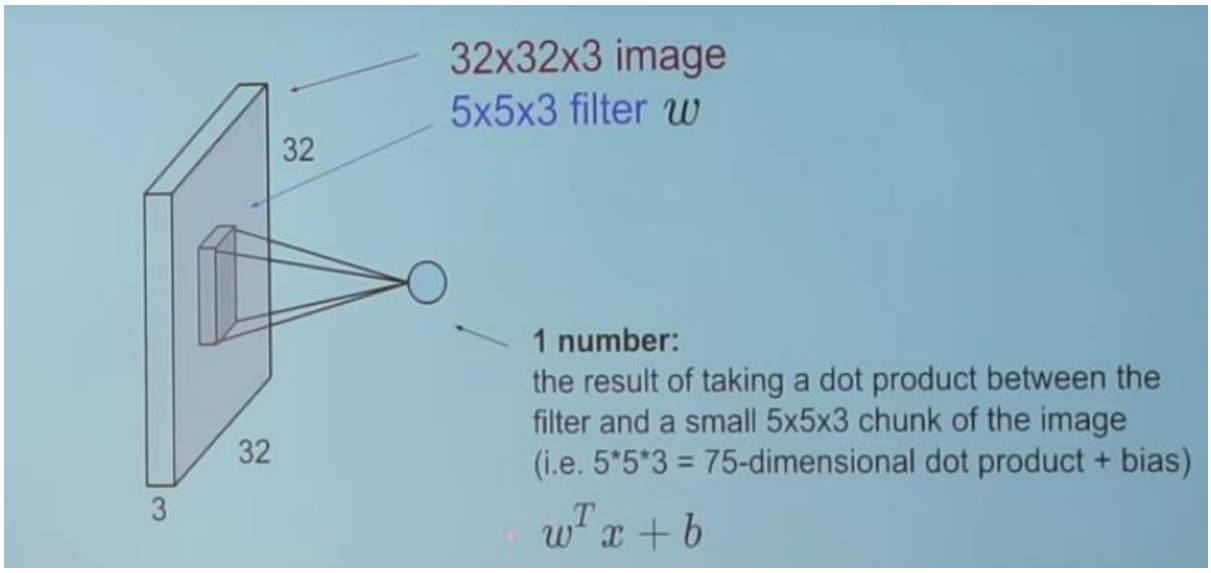
To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

However, after this shift/scale of activation outputs by some randomly initialized parameters, the weights in the next layer are no longer optimal. SGD (Stochastic gradient descent) undoes this normalization if it's a way for it to minimize the loss function.

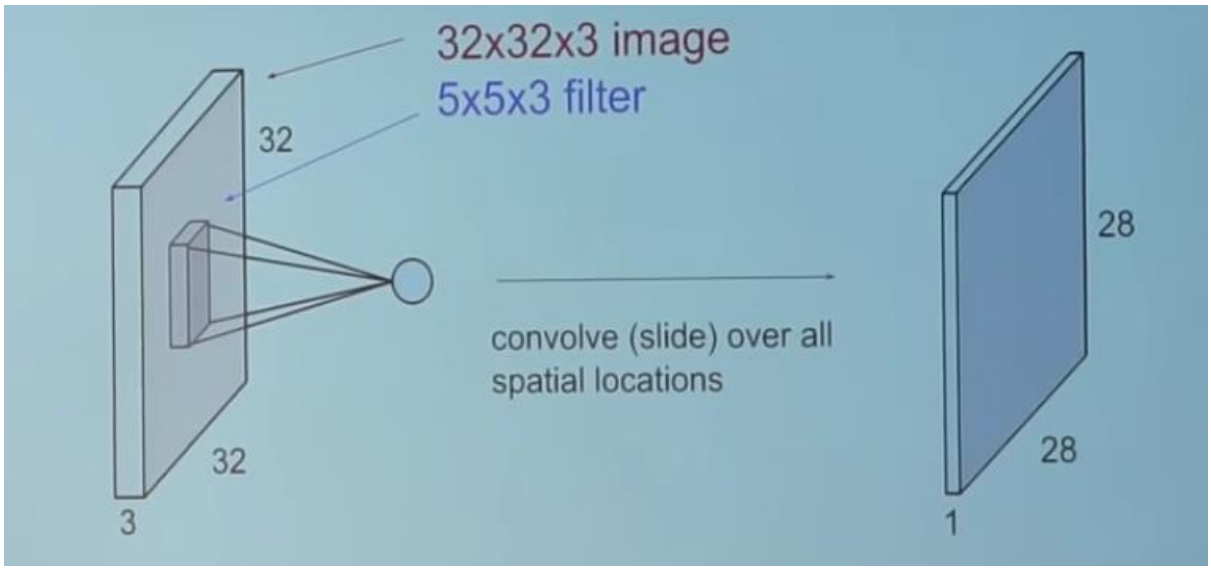
Consequently, batch normalization adds two trainable parameters to each layer, so the normalized output is multiplied by a "standard deviation" parameter (gamma) and add a "mean" parameter (beta). In other words, batch normalization lets SGD do the denormalization by changing only these two weights for each activation, instead of losing the stability of the network by changing all the weights.

Q9) What is convolution? Explain with a diagram & formula ?

We take the 5*5*3 filter and slide it over the complete image and along the way take the dot product between the filter and chunks of the input image.



For every dot product taken, the result is a scalar.



The convolution of f and g is written f∗g, using an asterisk. It is defined as the integral of the product of the two functions after one is reversed and shifted. As such, it is a particular kind of integral transform:

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau.$$

ACV-Question Bank

Q10) What is padding? Why is it required? (Formula mandatory)

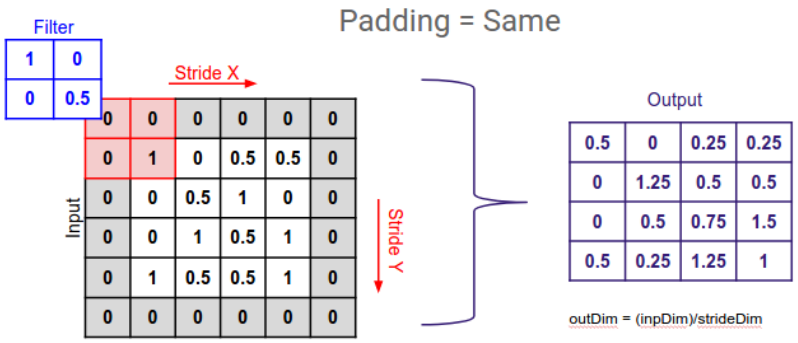
If you take a grey scale image, the pixel in the corner will only get covers one time but if you take the middle pixel it will get covered more than once basically what does that means is we have more info on that middle pixel so these are the two main downsides

- Shrinking outputs
- Loosing information on corners of the image

To overcome this, we can introduce Padding to an image. Padding is an additional layer that we can add to the border of an image. For an example see the figure below there one more layer added to the 4*4 image and now it has converted in to 5*5 image. So now there is more frame that covers the edge pixels of an image cool. More info more accuracy that's how neural net works so we have more info now we can get more accuracy done.

We have two options o padding:

- Pad the picture with zeros (zero-padding) so that it fits
- Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.



Q11) What is pooling? Why is it required? Explain atleast 3 methods. (Formula mandatory)

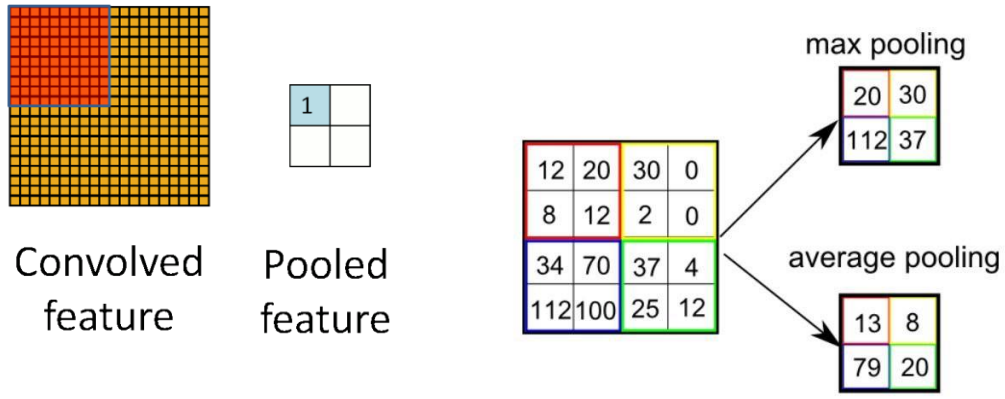
After obtaining features using convolution, we would next like to use them for classification. In theory, one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. Consider for instance images of size 96x96 pixels, and suppose we have learned 400 features over 8x8 inputs. Each convolution results in an output of size $(96-8+1)*(96-8+1)=7921$, and since we have 400 features, this results in a vector of $892*400=3,168,400$

features per example. Learning a classifier with inputs having 3+ million features can be unwieldy, and can also be prone to over-fitting.

To address this, first recall that we decided to obtain convolved features because images have the "stationarity" property, which implies that features that are useful in one region are also likely to be useful for other regions. Thus, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. For example, one could compute the mean (or max) value of a particular feature over a region of the image. These summary statistics are much lower in dimension (compared to using all of the extracted features) and can also improve results (less over-fitting). We aggregation operation is called this operation "'pooling"', or sometimes "'mean pooling"' or "'max pooling"' (depending on the pooling operation applied).

ACV-Question Bank

The following image shows how pooling is done over 4 non-overlapping regions of the image.



Pooling for Invariance

If one chooses the pooling regions to be contiguous areas in the image and only pools features generated from the same (replicated) hidden units. Then, these pooling units will then be "'translation invariant"'. This means that the same (pooled) feature will be active even when the image undergoes (small) translations. Translation-invariant features are often desirable; in many tasks (e.g., object detection, audio recognition), the label of the example (image) is the same even when the image is translated. For example, if you were to take an MNIST digit and translate it left or right, you would want your classifier to still accurately classify it as the same digit regardless of its final position.

Formal description: Formally, after obtaining our convolved features as described earlier, we decide the size of the region, say m×n to pool our convolved features over. Then, we divide our convolved features into disjoint m×n regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled convolved features. These pooled features can then be used for classification.

Max Pooling: Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.

One interesting property of max pooling is that it has a set of hyperparameters but it has no parameters to learn. There's actually nothing for gradient descent to learn. Once you fix f and s, it's just a fixed computation and gradient descent doesn't change anything.

The size of the output : $\left[\frac{n+2p-f}{s}-1\right]$

max pooling is used much more often than average pooling

Average Pooling: Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.

Average pooling is used in case of very deep in a neural network.

Global Pooling: Global pooling reduces each channel in the feature map to a single value. Thus, an nh x nw x nc feature map is reduced to 1 x 1 x nc feature map. This is equivalent to using a filter of dimensions nh x nw i.e. the dimensions of the feature map. Further, it can be either global max pooling or global average pooling.

Q12) Explain both Vanishing Gradient and Exploding Gradient ?

Vanishing Gradient:

It occurs when we try to train a neural network model using gradient based optimisation techniques.

Vanishing Gradient Problem was actually a major problem few years back to train a Deep neural Network Model due to the long training process and the degraded accuracy of the Model.

When we do Back-propagation i.e moving backward in the Network and calculating gradients of loss(Error) with respect to the weights , the gradients tends to get smaller and smaller as we keep on moving backward in the Network. This means that the **neurons** in the **Earlier** *layers learn very slowly* as compared to the neurons in the later layers in the Hierarchy. The Earlier layers in the network are slowest to train.

Multi-level hierarchy: This technique pretrains one layer at a time, and then performs backpropagation for fine tuning.

Residual networks: The technique introduces bypass connections that connect layers further behind the preceding layer to a given layer. This allows gradients to propagate faster to deep layers before they can be attenuated to small or zero values

Rectified linear units (ReLUs): When using rectified linear units, the typical sigmoidal activation functions used for node output is replaced with with a new function: f(x) = max(0, x). This activation only saturates on one direction and thus are more resilient to the vanishing of gradients.

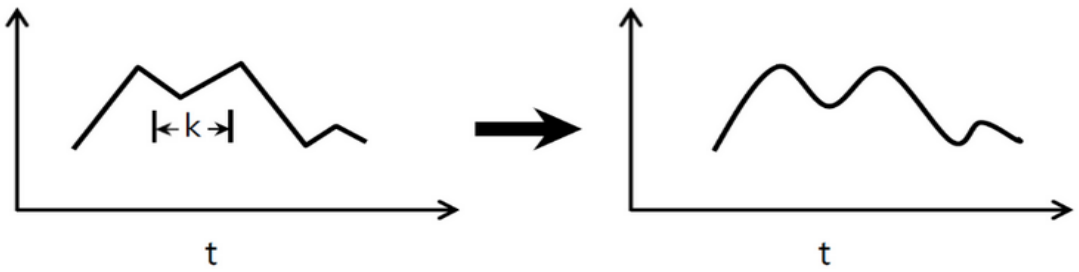
Exploding gradient: In machine learning, the exploding gradient problem is an issue found in training artificial neural networks with gradient-based learning methods and backpropagation. An artificial neural network is a learning algorithm, also called neural network or neural net, that uses a network of functions to understand and translate data input into a specific output. This type of learning algorithm is designed to mimic the way neurons function in the human brain. Exploding gradients are

a problem when large error gradients accumulate and result in very large updates to neural network model weights during training. Gradients are used during training to update the network weights, but when the typically this process works best when these updates are small and controlled. When the magnitudes of the gradients accumulate, an unstable network is likely to occur, which can cause poor predicition results or even a model that reports nothing useful what so ever. There are methods to fix exploding gradients, which include gradient clipping and weight regularization, among others.

Exploding gradients can cause problems in the training of artificial neural networks. When there are exploding gradients, an unstable network can result and the learning cannot be completed. The values of the weights can also become so large as to overflow and result in something called NaN values. NaN values, which stands for not a number, are values that represent an undefined or unrepresentable values. It is useful to know how to identify exploding gradients in order to correct the training.

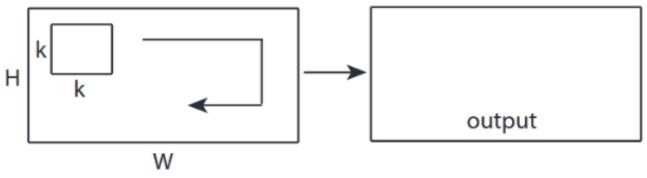
Q13) Explain uses of 1D, 2D and 3D convolutions ?

1D convolutions



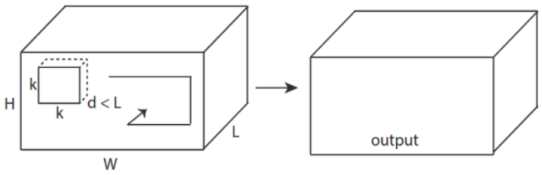
- just 1-direction (time-axis) to calculate conv
- input = [W], filter = [k], output = [W]
- ex) input = [1,1,1,1,1], filter = [0.25,0.5,0.25], output = [1,1,1,1,1]
- output-shape is 1D array

2D convolutions



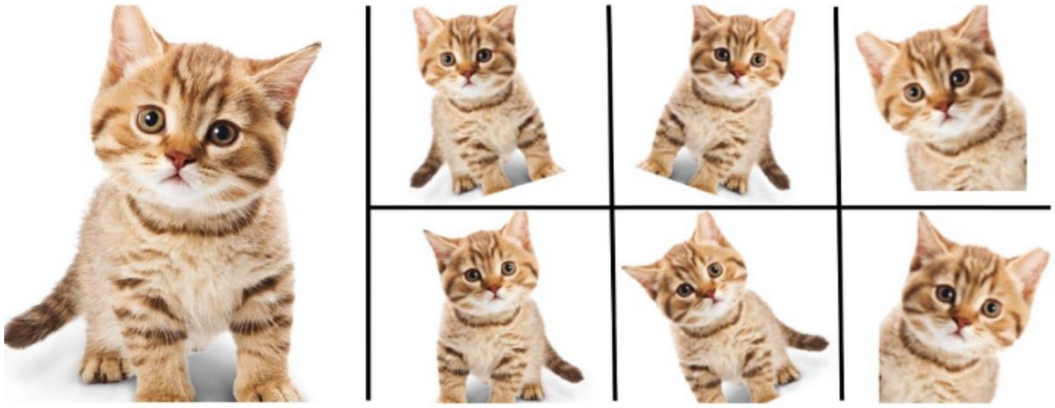
- **2**-direction (x,y) to calculate conv
- output-shape is **2D** Matrix
- input = [W, H], filter = [k,k] output = [W,H]

3D convolutions



- 3-direction (x,y,z) to calcuate conv
- output-shape is 3D Volume
- input = [W,H,L], filter = [k,k,d] output = [W,H,M]
- d < L is important! for making volume output
- example) C3D

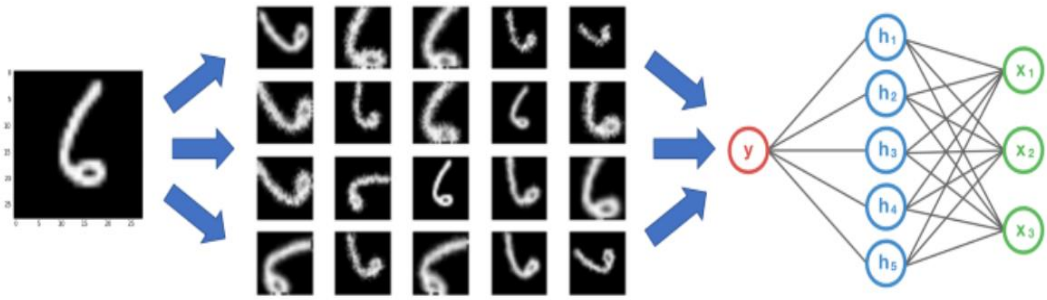
Q14) What is Data Augmentation?



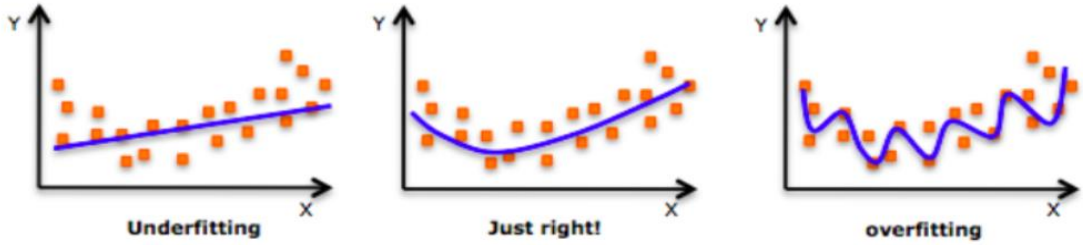
Data augmentation is a strategy that enables practitioners to significantly increase the diversity of data available for training models, without actually collecting new data.

Data augmentation techniques such as cropping, padding, and horizontal flipping are commonly used to train large neural networks. However, most approaches used in training neural networks only use basic types of augmentation. While neural network architectures have been investigated in depth, less focus has been put into discovering strong types of data augmentation and data augmentation policies that capture data invariances.

A convolutional neural network that can robustly classify objects even if its placed in different orientations is said to have the property called **invariance**. More specifically, a CNN can be invariant to **translation, viewpoint, size** or **illumination** (Or a combination of the above).

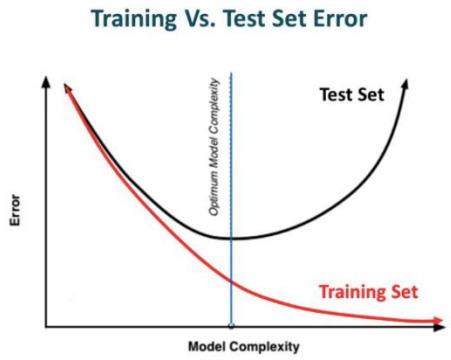


Q15) Explain atleast 2 Regularization techniques for DL. (Dropout and Data Augmentation) ?

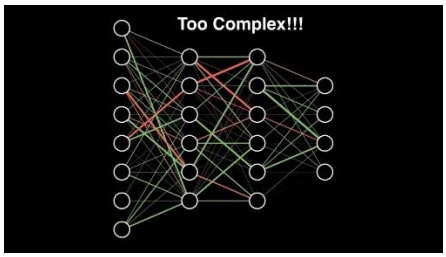


ACV-Question Bank

As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, which ultimately results in poor performance on the unseen data. In other words, while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't. This is shown in the image below.



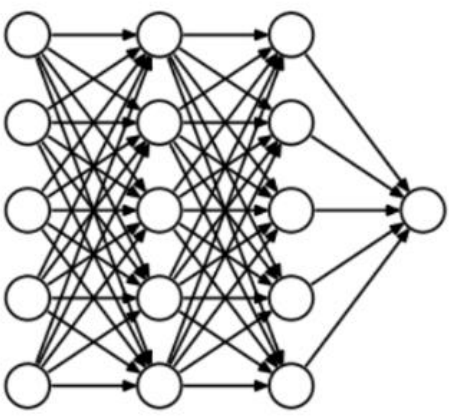
If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting.

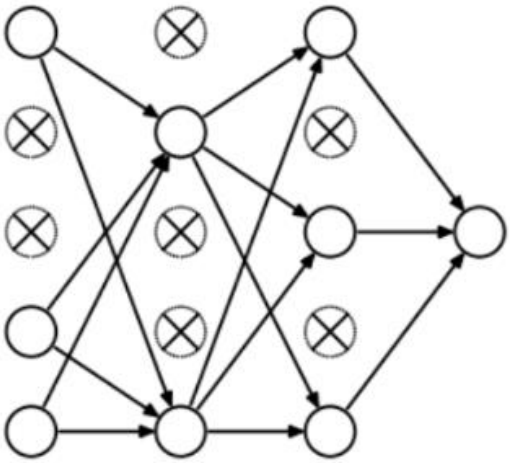


Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.

Dropout: This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning. To understand dropout, let's say our neural network structure is akin to the one shown below:

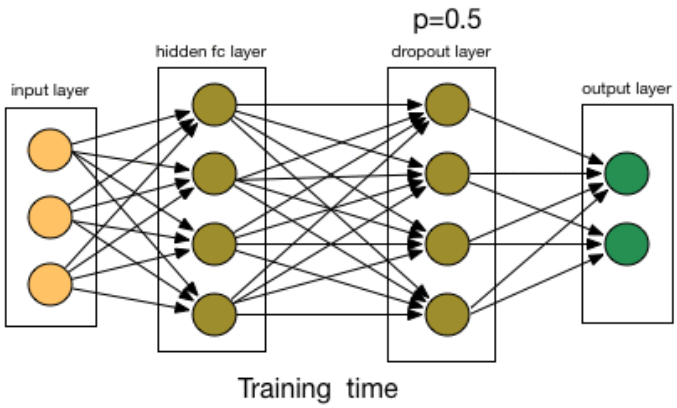
So what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below.





So each iteration has a different set of nodes and this results in a different set of outputs. **It can also be thought of as an ensemble technique in machine learning.** Ensemble models usually perform better than a single model as they capture more randomness. Similarly, dropout also performs better than a normal neural network model.

This probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function. As seen in the image above, dropout can be applied to both the hidden layers as well as the input layers.



Due to these reasons, dropout is usually preferred when we have a large neural network structure in order to introduce more randomness. In *keras*, we can implement dropout using the keras core layer. Below is the python code for it:

```
from keras.layers.core import Dropout

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])
```

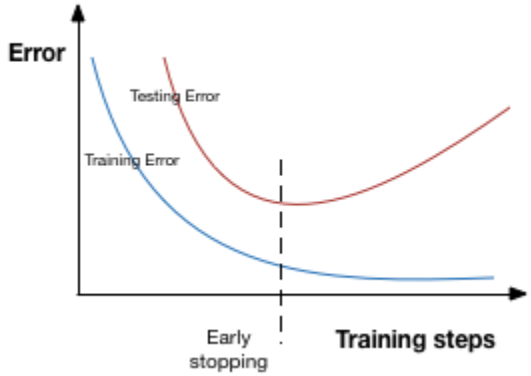
As you can see, we have defined 0.25 as the probability of dropping. We can tune it further for better results using the grid search method.

Data Augmentation: The simplest way to reduce overfitting is to increase the size of the training data. In machine learning, we were not able to increase the size of training data as the labeled data was too costly. But, now let's consider we are dealing with images. In this case, there are a few ways of increasing the size of the training data – rotating the image, flipping, scaling, shifting, etc. In the below image, some transformation has been done on the handwritten digits dataset.

This technique is known as data augmentation. This usually provides a big leap in improving the accuracy of the model. *It can be considered as a mandatory trick in order to improve our predictions.* In *keras*, we can perform all of these transformations using ImageDataGenerator. It has a big list of arguments which you you can use to pre-process your training data. Below is the sample code to implement it.

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

Early stopping: Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model. This is known as early stopping.



In the above image, we will stop training at the dotted line since after that our model will start overfitting on the training data. In keras, we can apply early stopping using the callbacks function. Below is the sample code for it.

```
from keras.callbacks import EarlyStopping

EarlyStopping(monitor='val_err', patience=5)
```

Here, monitor denotes the quantity that needs to be monitored and '**val_err**' denotes the validation error. Patience denotes the number of epochs with no further improvement after which the training will be stopped. For better understanding, let's take a look at the above image again. After the dotted line, each epoch will result in a higher value of validation error. Therefore, 5 epochs after the dotted line (since our patience is equal to 5), our model will stop because no further improvement is seen.

Note: It may be possible that after 5 epochs (this is the value defined for **patience** in general), the model starts improving again and the validation error starts decreasing as well. Therefore, we need to take extra care while tuning this hyperparameter.

Q16) What is early stopping?

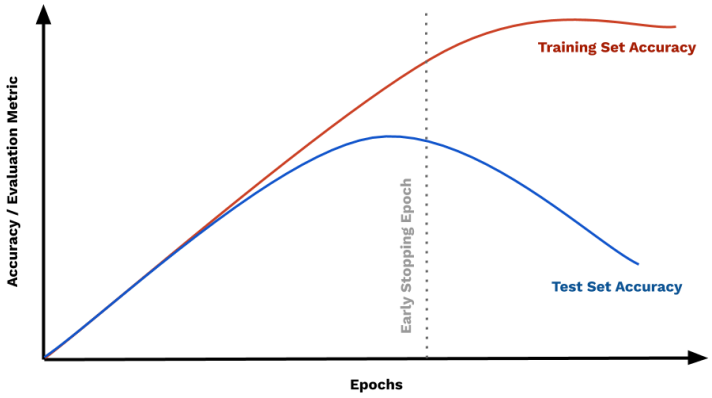
When training neural networks, numerous decisions need to be made regarding the settings (hyperparameters) used, in order to obtain good performance. Once such hyperparameter is the number of training epochs: that is, how many full passes of the data set (epochs) should be used? If we use too few epochs, we might underfit (i.e., not learn everything we can from

the training data); if we use too many epochs, we might overfit (i.e., fit the 'noise' in the training data, and not the signal). Early stopping attempts to remove the need to manually set this value. It can also be considered a type of regularization method (like L1/L2 weight decay and dropout) in that it can stop the network from overfitting.

The idea behind early stopping is relatively simple:

- Split data into training and test sets
- At the end of each epoch (or, every N epochs):
    - evaluate the network performance on the test set
    - if the network outperforms the previous best model: save a copy of the network at the current epoch
- Take as our final model the model that has the best test set performance

This is shown graphically below:

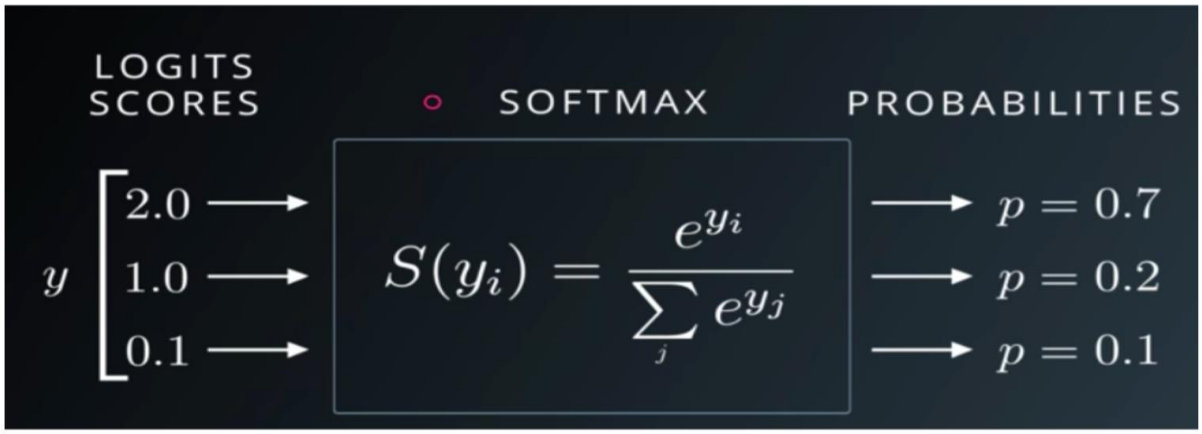


The best model is the one saved at the time of the vertical dotted line - i.e., the model with the best accuracy on the test set. Using DL4J's early stopping functionality requires you to provide a number of configuration options:

- A score calculator, such as the *DataSetLossCalculator*(JavaDoc, Source Code) for a Multi Layer Network, or *DataSetLossCalculatorCG* (JavaDoc, Source Code) for a Computation Graph. Is used to calculate at every epoch (for example: the loss function value on a test set, or the accuracy on the test set)
- How frequently we want to calculate the score function (default: every epoch)
- One or more termination conditions, which tell the training process when to stop. There are two classes of termination conditions:
    - Epoch termination conditions: evaluated every N epochs
    - Iteration termination conditions: evaluated once per minibatch
- A model saver, that defines how models are saved

Q17) What is Softmax? (Loss Function required)?

Activation function that turns logits into probabilities that sum to one. Softmax function outputs a vector that represents the probability distributions of a list of potential outcomes.



Logits are the raw scores output by the last layer of a neural network. Before activation takes place. Softmax is not a black box. It has two components: special number e to some power divide by a sum of some sort. y_i refers to each element in the logits vector y. Python and Numpy code will be used in this article to demonstrate math operations

Q18) Explain SGD, Momentum, Nesterov AG, Adagrad, Adadelta, Adam?

In Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration.

Although, using the whole dataset is really useful for getting to the minima in a less noisy or less random manner, but the problem arises when our datasets get really huge.

- **Local minima**
- **Slow convergence**
- **Different slopes**
- **Saddle points**
- **Gradient size & distributed training**

These problems can be solved by the following optimizers :

Gradient descent with Momentum: Compute an exponential weighted average of gradients and use them to update weights.

Aim : Reaching from the start point to the minima in a faster way.

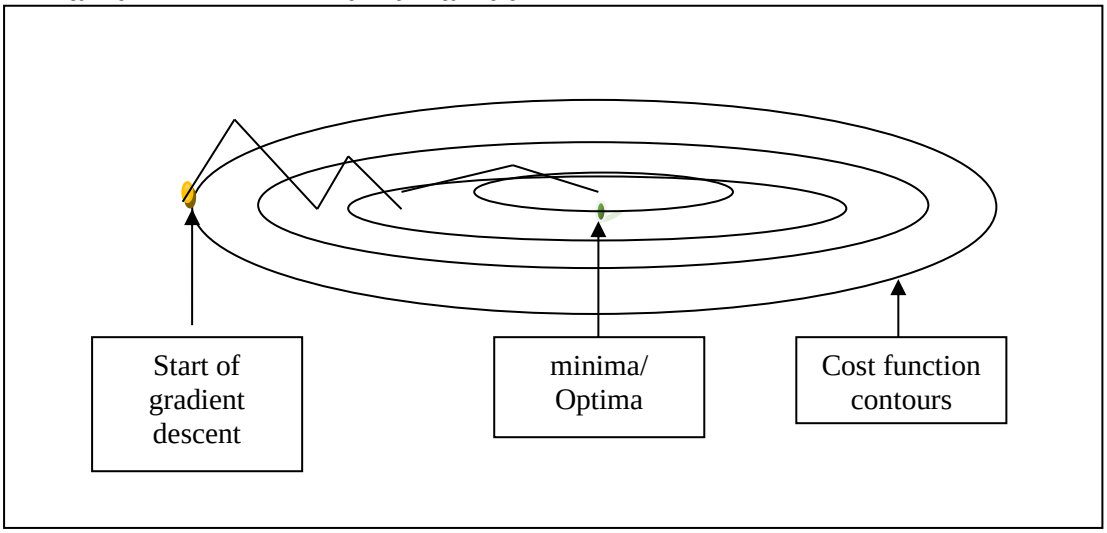
A very large learning rate in the vertical direction may result in overshooting and we may deviate from the path, a smaller learning in the horizontal direction rate may result in increase in the time to reach the minima.

Thus, it is necessary to choose a perfect learning rate, a lower learning rate in the vertical direction and a faster learning rate in the horizontal direction. This is when momentum comes into picture.

ACV-Question Bank

During backward propagation of standard method, we use dw and db to update our parameters W and b as follows:

$W = W - \alpha * dw$ $b = b - \alpha * db$



Steps involved in sgd with momentum::

On iteration t:

1. Compute dw, db on the current mini- batch / batch.

2. Compute :

$vdw = \beta * vdw + (1- \beta)dw$

$vd\theta = \beta * vd\theta + (1- \beta)d\theta$

$vdb = \beta * vdb + (1- \beta)db$

3. Updating:

$W = W - \alpha * vdw$

$b = b - \alpha * vdb$

where, α = Learning rate

Consider, a ball sliding downwards from the top of a valley. Let, the valley represent our gradient descent contours and the ball tracing the path from the starting point to minima. Then, the derivatives(dw, db) would be analogous to the acceleration of the ball and the momentum terms (vdw, vdb) would be the velocity of the ball. Friction that prevents the ball from speeding up (overshooting) beyond the limits is our hyperparameter β in this case.

Advantages:

1. Faster than standard methods.

2. Updating weights using exponential weighted average would result in the oscillations in the vertical direction to average out tending to zero. damping out the oscillations in vertical directions would result in curve smoothening and also mean slower learning rate in vertical direction.

3. In horizontal directions all the derivatives point towards right(minima) so the average in the horizontal direction is large resulting in straight forward and faster path towards minima.

ACV-Question Bank

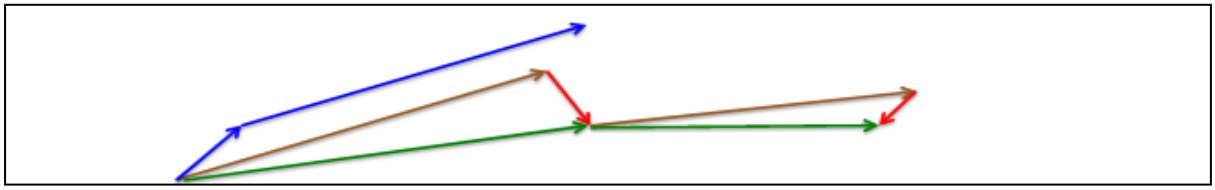
Nesterov accelerated gradient (NAG)

In sgd with momentum we took an example of a ball moving down the valley but this ball din't have a notion of where it is going if it had it could slow down before the valley slopes up again(on reaching minima).This is what NAG does.

The standard momentum method first computes the gradient at the current location and then takes a big jump in the direction of the updated accumulated gradient.

Steps involved in NAG

1. First make a big jump in the direction of the previous accumulated gradient.
2. Measure the gradient where you end up and make a correction.



Brown vector = jump, red vector = correction, green vector = accumulated gradient, blue vector = standard momentum

On iteration t:

$V_t = \beta * V_{t-1} + \eta \nabla_\theta J(\theta - \beta * V_{t-1})$

$\theta = \theta - V_t$

ADAGRAD

An algorithm that adjusts the learning rate according to the gradient value of the independent variable in each dimension to eliminate problems caused when a unified learning rate has to adapt to all dimensions.

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the learning rate.

The Adagrad algorithm uses the cumulative variable st obtained from a square by element operation on the mini-batch stochastic gradient gt. At time step 0, Adagrad initializes each element in s0 to 0. At time step t, we first sum the results of the square by element operation for the mini-batch gradient gt to get the variable st

$s_t \leftarrow s_{t-1} + g_t \odot g_t,$

Here, $\odot$ is the symbol for multiplication by element. Next, we re-adjust the learning rate of each element in the independent variable of the objective function using element operations:

$x_t \leftarrow x_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t,$

Here, η is the learning rate . Here, the square root, division, and multiplication operations are all element operations. Each element in the independent variable of the objective function will have its own learning rate after the operations by elements.We should emphasize that the cumulative variable st produced by a square by element operation on the mini-batch stochastic gradient is part of the learning rate denominator. Therefore, if an element in the

independent variable of the objective function has a constant and large partial derivative, the learning rate of this element will drop faster. On the contrary, if the partial derivative of such an element remains small, then its learning rate will decline more slowly. However, since st accumulates the square by element gradient, the learning rate of each element in the independent variable declines (or remains unchanged) during iteration. Therefore, when the learning rate declines very fast during early iteration, yet the current solution is still not desirable, Adagrad might have difficulty finding a useful solution because the learning rate will be too small at later stages of iteration.

- Adagrad constantly adjusts the learning rate during iteration to give each element in the independent variable of the objective function its own learning rate.
- When using Adagrad, the learning rate of each element in the independent variable decreases (or remains unchanged) during iteration.

Disadvantages of adagrad

1) the continual decay of learning rates throughout training, and
2) the need for a manually selected global learning rate.

ADADELTA

It helps improve the chances of finding useful solutions at later stages of iteration, which is difficult to do when using the Adagrad algorithm for the same purpose. The interesting thing is that there is no learning rate hyperparameter in the Adadelta algorithm.

Like RMSProp, the Adadelta algorithm uses the variable st, which is an EWMA on the squares of elements in mini-batch stochastic gradient gt. At time step 0, all the elements are initialized to 0. Given the hyperparameter $0 \le \rho < 1$ (counterpart of γ in RMSProp), at time step $t > 0$, compute using the same method as RMSProp:

$$s_t \leftarrow \rho s_{t-1} + (1-\rho) g_t \odot g_t.$$

Unlike RMSProp, Adadelta maintains an additional state variable, Δxtthe elements of which are also initialized to 0 at time step 0. We use Δxt−1to compute the variation of the independent variable:

$$g'_t \leftarrow \sqrt{\Delta x_{t-1} + \epsilon} \odot g_t,$$

Here, ϵis a constant added to maintain the numerical stability, such as 10−5. Next, we update the independent variable:

$$x_t \leftarrow x_{t-1} - g'_t.$$

Finally, we use Δx to record the EWMA on the squares of elements in g′, which is the variation of the independent variable.

$$\Delta x_t \leftarrow \rho \Delta x_{t-1} + (1-\rho) g'_t \odot g'_t.$$

As we can see, if the impact of ϵis not considered here, Adadelta differs from RMSProp in its replacement of the hyperparameter η with √Δxt−1

ADAMAX

It is a variant of Adam based on the infinity norm. In Adam, the update rule for individual weights is to scale their gradients inversely proportional to a(scaled)L2norm of their individual current and past gradients. We can generalize theL2norm based update rule to aLp norm based update rule. Such variants become numerically unstable for large p. However, in the special case where we let p→ ∞, a surprisingly simple and stable algorithm emerges. Let, in case of the Lp norm, the step size at time t be inversely proportional to vt1/p, where:

vt = β2p vt−1+ (1−β2p) |gt|p

= (1−β2p)$\sum_{i=1}^{t} \beta 2$ p(t−i)·|gi|p

Note that the decay term is here equivalently parameterized as β2p instead of β2. Now let p→ ∞,and define ut= limp→∞(vt)1/p, then:

ut= limp→∞(vt)1/p = limp→∞((1−β2p) $\sum_{i=1}^{t} \beta 2$p(t−i)·|gi|p)1/p

= max(β2t−1|g1|,β2t−2|g2|,...,β2|gt−1|,|gt|)

Which corresponds to the remarkably simple recursive formula

ut= max(β2·vt−1,|gt|)

with initial valueu0= 0. Note that, conveniently enough, we don't need to correct for initialization bias in this case. Also note that the magnitude of parameter updates has a simpler bound with AdaMax than Adam, namely:|Δt|≤α

Adam

adds **momentum** to AdaDelta, further focusing progress in the direction of steepest descent.

Adam stands for Adaptive Moment Estimation. Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta ,Adam also keeps an exponentially decaying average of past gradients M(t), similar to momentum:

M(t) and V(t) are values of the first moment which is the Mean and the second moment which is the uncentered variance of the gradients respectively.

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The formulas for the first Moment(mean) and the second moment (the variance) of the Gradients. Then the final formula for the Parameter update is —

$$\theta_{t+1} = \theta_t - \frac{\eta \widehat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

The values for β1 is 0.9 , 0.999 for β2, and (10 x exp(-8)) for ϵ.

Adam works well in practice and compares favourably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quiet Fast and efficient and also it rectifies every problem that is faced in other optimization techniques such as vanishing Learning rate , slow convergence or High variance in the parameter updates which leads to fluctuating Loss function

Q19) What is cyclical learning rate?

A technique to set and change and tweak *LR* during training. This methodology aims to train neural network with a LR that changes in a cyclical way for each batch, instead of a non-cyclic LR that is either constant or changes on every epoch. The learning rate schedule varies between two bounds.

When using a cyclical LR, we have to calculate two things :
1) The bounds between which the learning rate will vary — **base_lr** and **max_lr**.
2) The step_size — in how many epochs the learning rate will reach from one bound to the other.

Why it works?

We have always learnt that we should keep decreasing LR as training progresses so that we converge with time.

In CLR, we vary the LR between a lower and higher threshold. The logic is that periodic higher learning rates within each epoch helps to come out of any saddle points or local minima if it encounters into one. If saddle point happens to be an elaborated plateau, lower learning rates will probably never generate enough gradient to come out of it, resulting in difficulty in minimising the loss.

Objective: Pick a learning rate and change it on each iteration(batch) to make the training process performant — which means -:

1. Achieve the maximum possible accuracy in order to get best prediction results.
2. Speed up the training process by achieving above in minimum number of epochs .

Important Terms

Epoch: One epoch is completed when an entire dataset is passed forward and backward only once through the neural network.

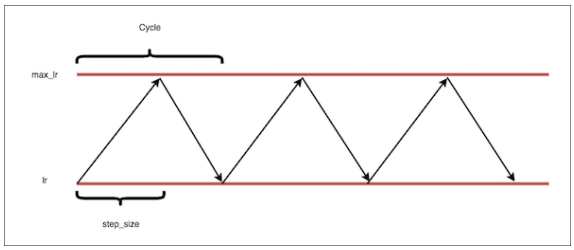
Batch Size: Number of training examples to utilise in one iteration.

Batch or Iteration: A training set of 1000 examples, with a batch size of 20 will take 50 iterations/batches to complete one epoch.

ACV-Question Bank

Cycle: Number of iterations we want for our learning rate to go from lower bound to upper bound, and then back to lower bound.

Step size: Number of iterations to complete half of a cycle.



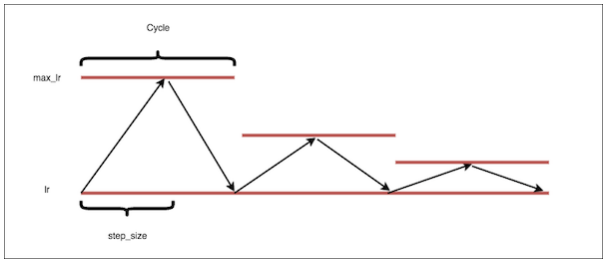
Setting base_lr and max_lr

The loss plot will see a decrease in loss as we increase the learning rate, but will start increasing again at a point. Note the LR at which loss starts to decrease, and also the LR when it starts stagnating. These are good points to set as *base_lr* and *max_lr.*

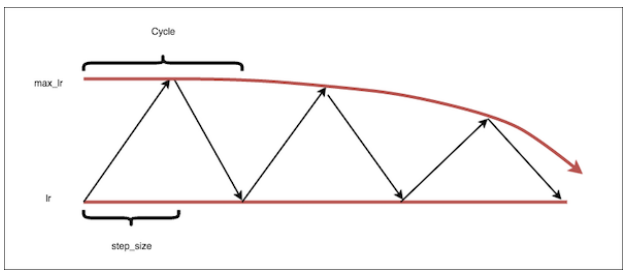
Alternatively, you can note the LR where accuracy peaks, and use that as *max_lr*. Set *base_lr* as 1⁄3 or 1⁄4 of this.

Variations of CLR: Other than triangular profile used above, Lesley Smith also suggested some other forms of CLR.

Triangular2 : Here the *max_lr* is halved after every cycle.



Exponential Range : Here *max_lr* is reduced exponentially with each iteration.



Conclusion

Cyclical Learning Rate is an amazing technique setting and controlling learning rates for training a neural network to achieve maximum accuracy, in a very efficient way.

Q20) What is SGD with warm restarts?

Restart techniques are common in gradient-free optimization to deal with multimodal functions. Partial warm restarts are also gaining popularity in gradient-based optimization to improve the rate of convergence in accelerated gradient schemes to deal with ill-conditioned functions. A simple warm restart technique for stochastic gradient descent to improve its anytime performance when training deep neural networks.

The existing restart techniques can also be used for stochastic gradient descent if the stochasticity is taken into account. Since gradients and loss values can vary widely from one batch of the data to another, one should denoise the incoming information: by considering averaged gradients and losses, e.g., once per epoch, the above-mentioned restart techniques can be used again. In this work, we consider one of the simplest warm restart approaches. We simulate a new warmstarted run / restart of SGD once Ti epochs are performed, where i is the index of the run. Importantly, the restarts are not performed from scratch but emulated by increasing the learning rate ηt whiletheoldvalueofxt isusedasaninitialsolution. Theamountofthisincreasecontrolstowhich extent the previously acquired information (e.g., momentum) is used. Within the i-th run, we decay the learning rate with a cosine annealing for each batch as follows:

ηt = ηimin + 1 2

(ηimax −ηimin)(1 + cos(Tcur Ti

π)), (5)

where ηimin and ηimax are ranges for the learning rate, and Tcur accounts for how many epochs have been performed since the last restart. Since Tcur is updated at each batch iteration t, it can take discredited values such as 0.1, 0.2, etc. Thus, ηt = ηimax when t = 0 and Tcur = 0. Once Tcur = Ti, the cos function will output−1 and thus ηt = ηimin. The decrease of the learning rate is shown in Figure 1 for fixed Ti = 50, Ti = 100 and Ti = 200; note that the logarithmic axis obfuscates the typical shape of the cosine function. In order to improve anytime performance, we suggest an option to start with an initially small Ti and increase it by a factor of Tmult at every restart (see, e.g., Figure 1 for T0 = 1,Tmult = 2 and T0 = 10,Tmult = 2). It might be of great interest to decrease ηimax and ηimin at every new restart. However, for the sake of simplicity, here, we keep ηimax and ηimin the same for every i to reduce the number of hyperparameters involved. Since our simulated warm restarts (the increase of the learning rate) often temporarily worsen performance, we do not always use the last xt as our recommendation for the best solution (also called the incumbent solution). While our recommendation during the first run (before the first restart) is indeed the last xt, our recommendation after this is a solution obtained at the end of the last performed run at ηt = ηimin. We emphasize that with the help of this strategy, our method does not require a separate validation data set to determine a recommendation.

Q21) Explain L2 regularization with general loss formula?

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

An L2-regularized version of the cost function used in SGD for NN

When we are using Stochastic Gradient Descent (SGD) to fit our network's parameters to the learning problem at hand, we take, at each iteration of the algorithm, a step in the solution space towards the gradient of the loss function $J(\vartheta; X, y)$ in respect to the network's parameters ϑ. Since the solution space of deep neural networks is very rich, this method of learning might overfit to our training data. This overfitting may result in significant generalization error and bad performance on unseen data (or test data, in the context of model development), if no counter-measure is used. Those counter-measures are called regularization techniques.

"Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."

— Ian Goodfellow

$$J(W; X, y) + \lambda \cdot ||W||^2$$

Equation 2 : Adding L^2 regularization to a cost function

where $\|\cdot\|$ is the L^2 norm. This is, indeed, the form we encounter in classical *Tikhonov regularization*.

Q22) Explain 4 different loss functions ?

A loss function or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a loss function.

Regression Losses: Mean Square Error/Quadratic Loss/L2 Loss (Euclidian Distance)

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

```
def MSE(yHat, y):
    return np.sum((yHat - y)**2) / y.size
```

Mean Absolute Error/L1 Loss (Manhattan Distance)

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n}$$

```
def L1(yHat, y):
    return np.sum(np.absolute(yHat - y)
```

ACV-Question Bank

Huber Loss

$$L_\delta = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\ |(y-\hat{y})| < \delta \\ \delta((y-\hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases}$$

It's less sensitive to outliers than the MSE as it treats error as square only inside an interval.

```
def Huber(yHat, y, delta=1.):
    return np.where(np.abs(y-yHat) < delta,.5*(y-yHat)**2 , delta*(np.abs(y-yHat)-0.5*delta))
```

Classification *Losses:*

Hinge Loss/Multi class SVM Loss

$$SVMLoss = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

```
def Hinge(yHat, y):
    return np.max(0, 1 - yHat * y)
```

Cross- Entropy Loss

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i))$$

```
def CrossEntropy(yHat, y):
    if y == 1:
      return -log(yHat)
    else:
      return -log(1 - yHat)
```

Kullback Leibler Loss

$$KL(P||Q) = \int p(x) log(\frac{p(x)}{q(x)}) dx$$

```
def KLDivergence(yHat, y):
    """
    :param yHat:
    :param y:
    :return: KLDiv(yHat || y)
    """
```

```
return np.sum(yHat * np.log((yHat / y)))
```

Q 23) UNDERFITTING and OVERFITTING

Underfitting:

A statistical model or a machine learning algorithm is said to have underfitting when it cannot capture the underlying trend of the data.

Underfitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough.

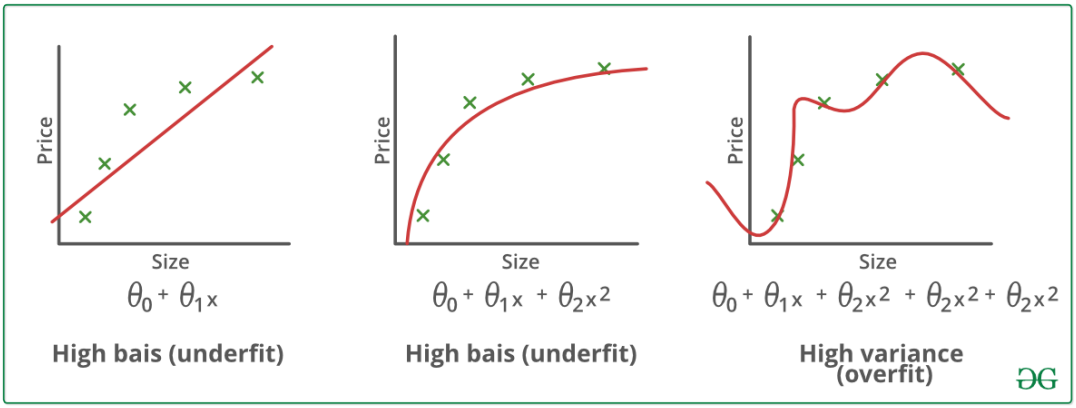
It usually happens when we have less data to build an accurate model and also when we try to build a linear model with a non-linear data.

In such cases the rules of the machine learning model are too easy and flexible to be applied on such a minimal data and therefore the model will probably make a lot of wrong predictions. Underfitting can be avoided by using more data and also reducing the features by feature selection.

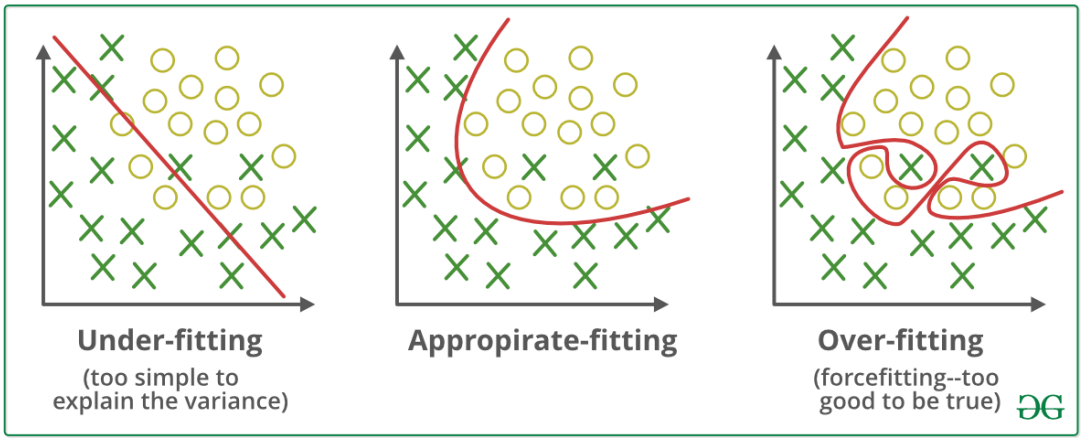
Overfitting:

A statistical model is said to be overfitted, when we train it with a lot of data (just like fitting ourselves in an oversized pants!). When a model gets trained with so much of data, it starts learning from the noise and inaccurate data entries in our data set. Then the model does not categorize the data correctly, because of too much of details and noise. The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore they can really build unrealistic models. A solution to avoid overfitting is using a linear algorithm if we have linear data or using the parameters like the maximal depth if we are using decision trees.

Bias:It gives us how closeness is our predictive model's to training data after averaging predict value. Generally algorithm has high bias which help them to learn fast and easy to understand but are less flexible. That loses it ability to predict complex problem, so it fails to explain the algorithm bias. This results in underfitting of our model.



Variance: It define as deviation of predictions, in simple it is the amount which tell us when its point data value change or a different data is use how much the predicted value will be affected for same model or for different model respectively. Ideally, the predicted value which we predict from model should remain same even changing from one training data-sets to another, but if the model has high variance then model predict value are affect by value of data-sets.



Q 24. What is model checkpointing?

Another word for model Checkpointing is Saving/Loading your model.

Deep learning models can take hours, days or even weeks to train.

If the run is stopped unexpectedly, you can lose a lot of work and there are many things to tweak and change over time.

MODEL CHECKPOINTING is a fault tolerance technique for long running processes, where a snapshot of the state of the system is taken in case of system failure. If there is a problem, not all is lost. The checkpoint may be used directly, or used as the starting point for a new run, picking up where it left off.

When training deep learning models, the checkpoint is the weights of the model. These weights can be used to make predictions as is, or used as the basis for ongoing training.

The Model Checkpoint callback class allows you to define where to checkpoint the model weights, how the file should name and under what circumstances to make a checkpoint of the model.

1- Best State

The ultimate goal of any learning project is to find the best model, the one the fits just right to the training set and generalizes well, So it makes sense you check every iteration if the model achieves a better score on your own metric and save it if so

2- Latest State

Like we said, training a model takes time.

And you may need to pause the training for any reason and continue training later without having to start over. Also, it's possible that you lose connection to the working environment.

So, you may need to save the latest state of your model every epoch of training, for you to be able to load it later and continue from where you were.

Q 25. Explain the following Architectures wrt:

- **Architecture Diagram**
- **Description of peculiar components (eg Residual Blocks, Inception Blocks, etc.)**
- **Loss Function**
- **VGG19, Resnet, Inception, Inception-Resnet, DenseNet, ResNext, Xception, SENet, MobileNet**
- **RCNN Family (4 Architectures), YOLO, SSD, RetinaNet, U-Net, FCN, Mask RCNN, GAN (Vanilla).**

VGG16 and VGG19

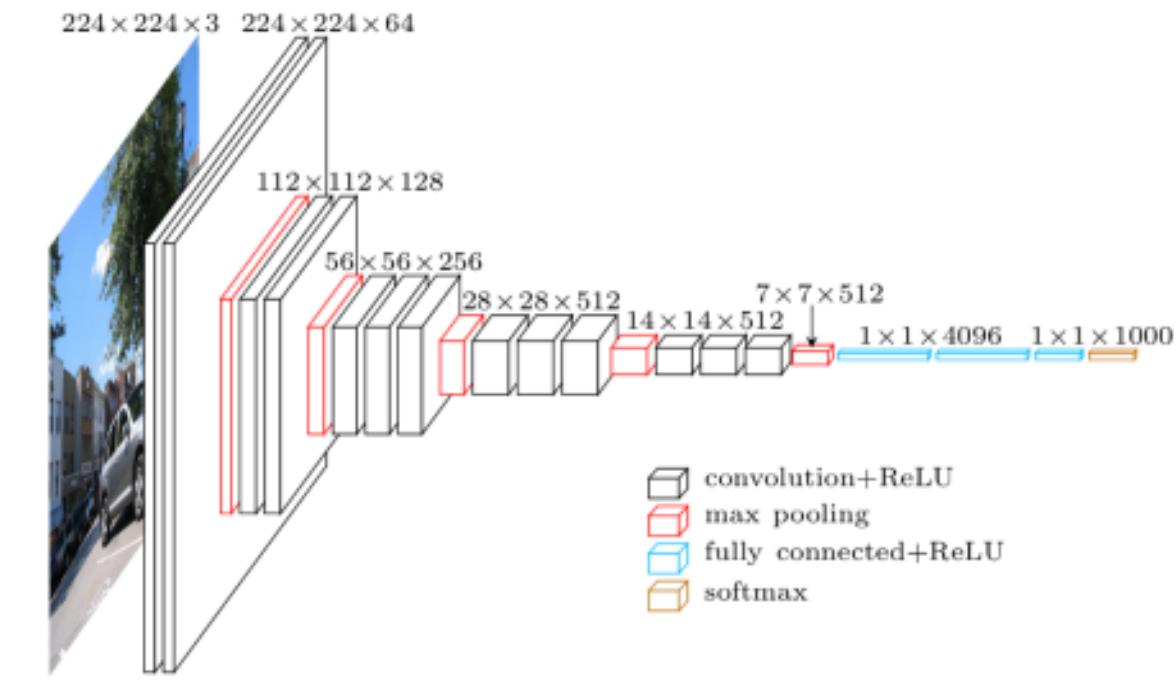


Figure 1: VGG architecture

The VGG network architecture was introduced by Simonyan and Zisserman in their 2014 paper, *Very Deep Convolutional Networks for Large Scale Image Recognition*.

This network is characterized by its simplicity, using only *3×3* convolutional layers stacked on top of each other in increasing depth. Reducing volume size is handled by max pooling. Two fully-connected layers, each with 4,096 nodes are then followed by a softmax classifier (above).

The smaller networks converged and were then used as *initializations* for the larger, deeper networks — this process is called ***pre-training***.

While making logical sense, pre-training is a very time consuming, tedious task, requiring an *entire network* to be trained **before** it can serve as an initialization for a deeper network.

Due to its depth and number of fully-connected nodes, VGG is over 533MB for VGG16 and 574MB for VGG19. This makes deploying VGG a tiresome task.

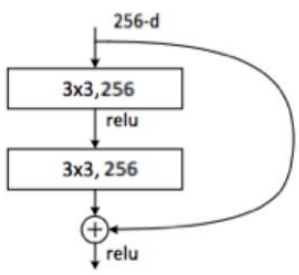
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

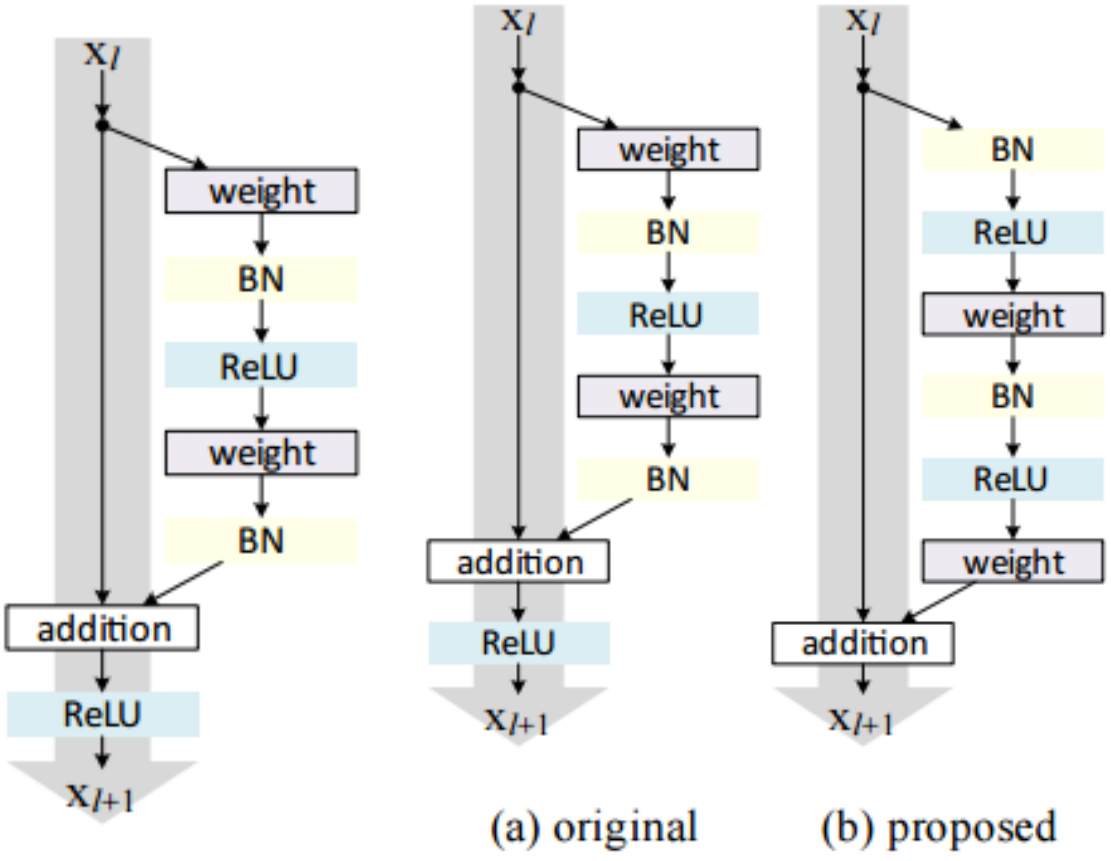
Figure: table of the VGG 16 and 19 created in 2019

ResNet

ResNet is a form of "exotic architecture" that relies on micro-architecture modules or 'network in network architectures'. First introduced by He et al. in their 2015 paper, Deep Residual Learning for Image Recognition, the ResNet architecture has become a seminal work, demonstrating that extremely deep networks can be trained using standard SGD through the use of residual Modules. A building block of a ResNet is called a **residual block** or **identity block**. A residual block is simply when the activation of a layer is fast-forwarded to a deeper layer in the neural network.

As you can see in the image above, the activation from a previous layer is being added to the activation of a deeper layer in the network. This, simple tweak allows training much deeper neural networks.

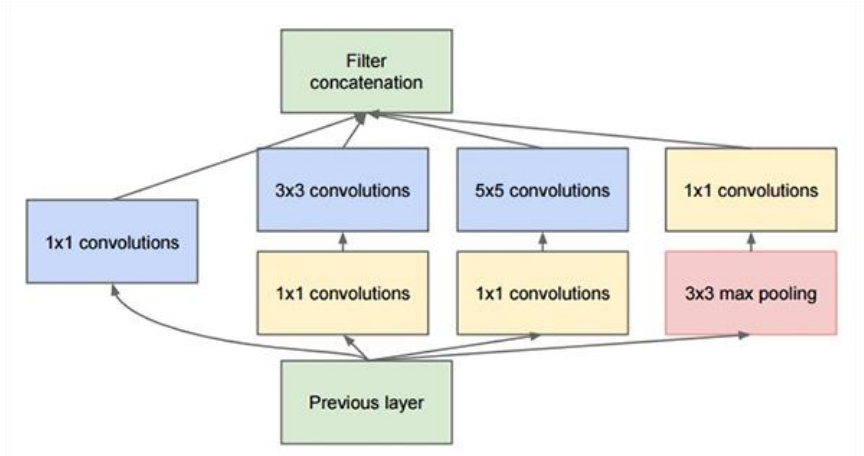




(Left) The original residual module. *(Right)* The updated residual module using pre-activation. Even though ResNet is much deeper than VGG16 and VGG19, the model size is actually substantially smaller due to the usage of global average pooling rather than fully-connected layers — this reduces the model size down to 102MB for ResNet50.

Inception V3

The "Inception" micro-architecture was first introduced by Szegedy et al. in their 2014 paper, Going Deeper with Convolutions:



ACV-Question Bank

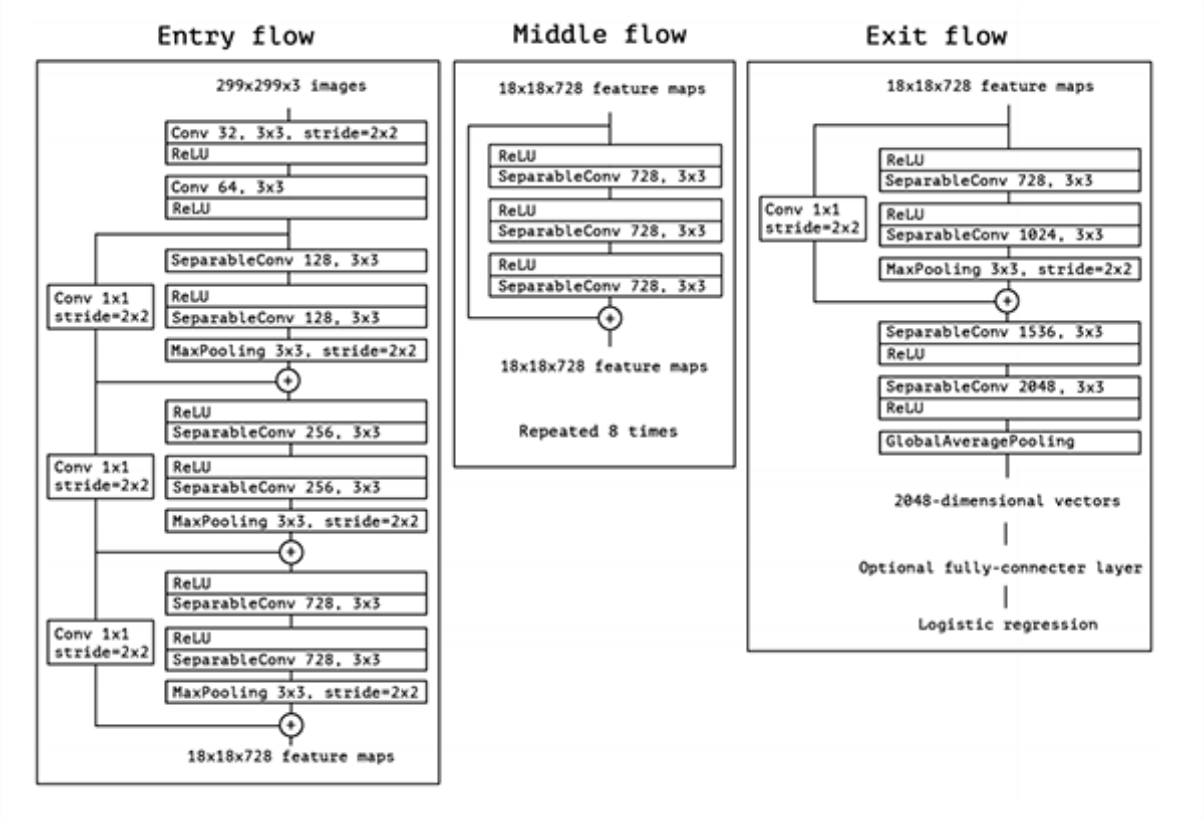
The goal of the inception module is to act as a “multi-level feature extractor” by computing 1×1, 3×3, and 5×5 convolutions within the same module of the network — the output of these filters are then stacked along the channel dimension and before being fed into the next layer in the network.

The Inception V3 architecture included in the Keras core comes from the later publication by Szegedy et al., Rethinking the Inception Architecture for Computer Vision (2015) which proposes updates to the inception module to further boost ImageNet classification accuracy. The weights for Inception V3 are smaller than both VGG and ResNet, coming in at 96MB

Xception

Xception was proposed by none other than François Chollet himself, the creator and chief maintainer of the Keras library.

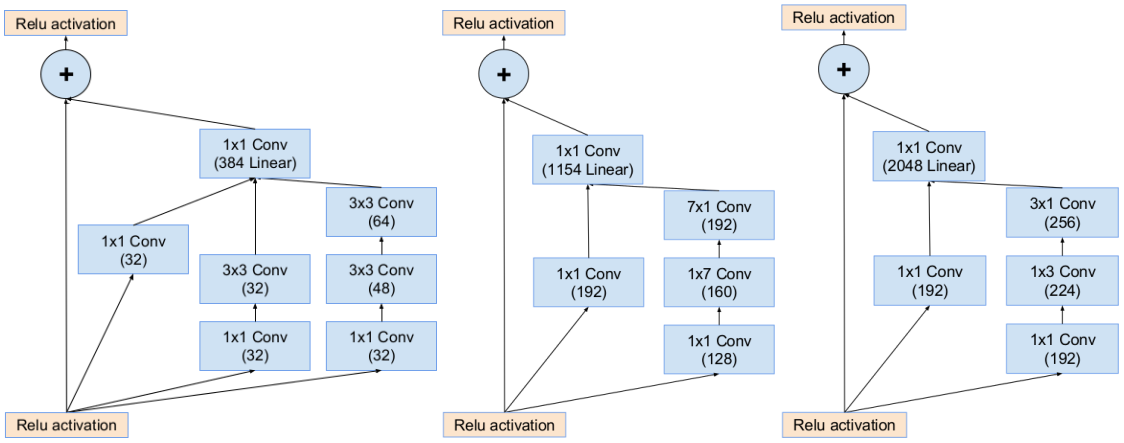
Xception is an extension of the Inception architecture which replaces the standard Inception modules with depthwise separable convolutions.



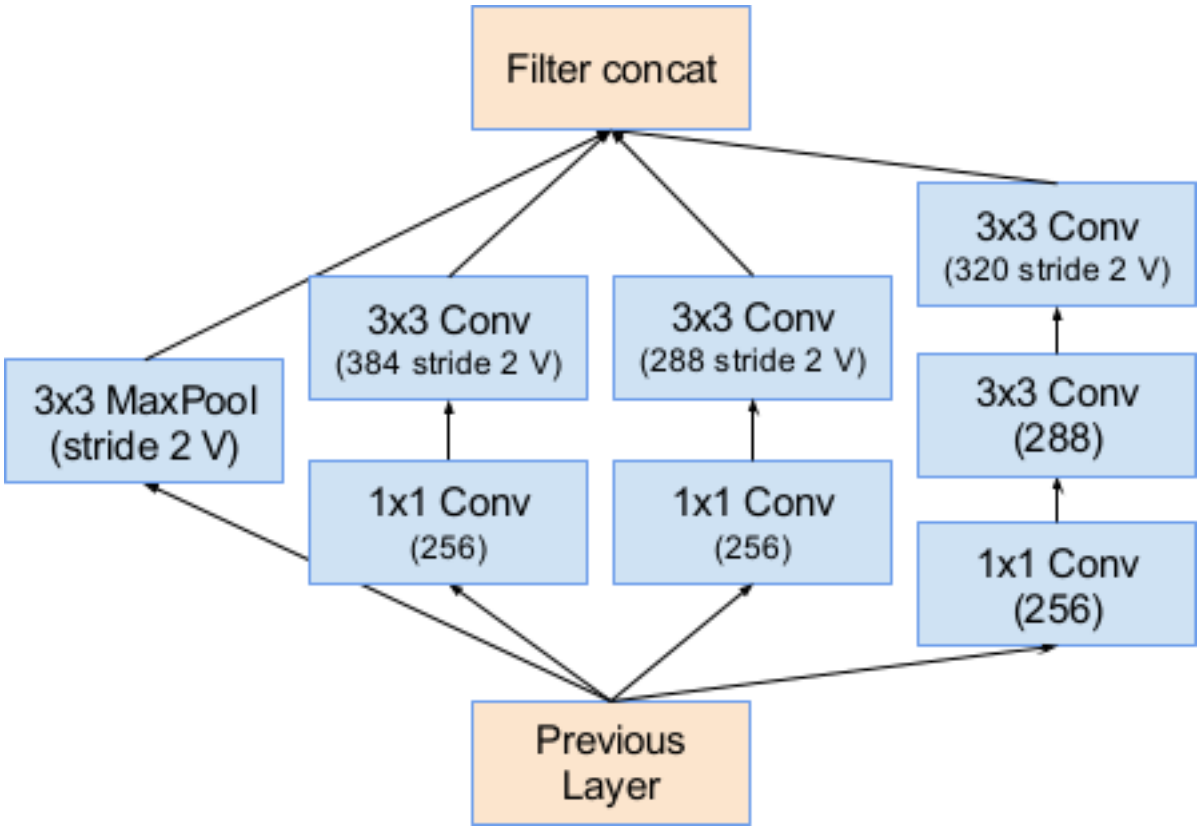
The original publication, Xception: Deep Learning with Depthwise Separable Convolutions can be found here. Xception sports the smallest weight serialization at only 91MB.

Inception-Resnet

ACV-Question Bank



Module A Module B Module C



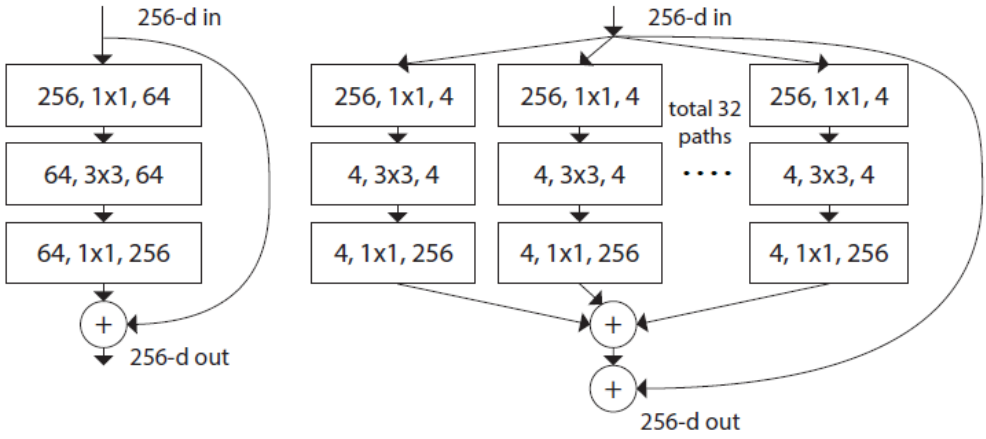
Inception-ResNet Architecture

Error function: RMSProb (SGD+Momentum apprently performed worse)

ResNeXt

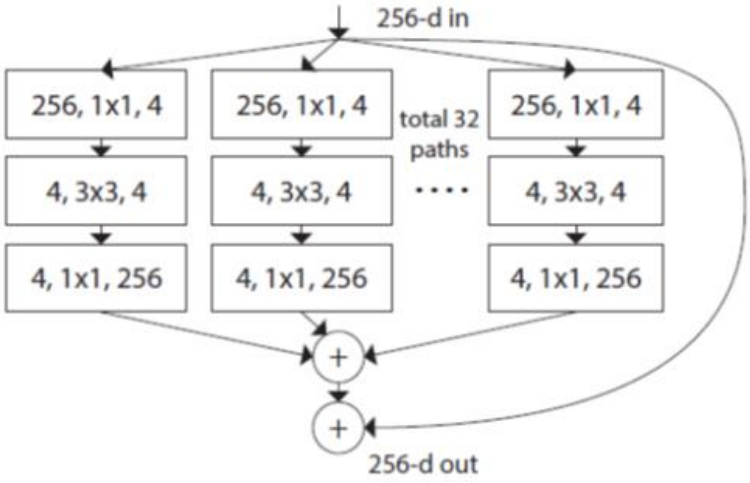
ResNeXt is a simple, highly modularized network architecture for image classification. The network is constructed by repeating a building block that aggregates a set of transformations with the same topology. The simple design results in a homogeneous, multi-branch architecture that has only a few hyper-parameters to set. This strategy exposes a new dimension, which we call “cardinality” (the size of the set of transformations), as an essential factor in addition to the dimensions of depth and width.

ACV-Question Bank



Residual Block in ResNet (Left), A Block of ResNeXt with Cardinality = 32 (Right)

New implementations in Resnext by facebook

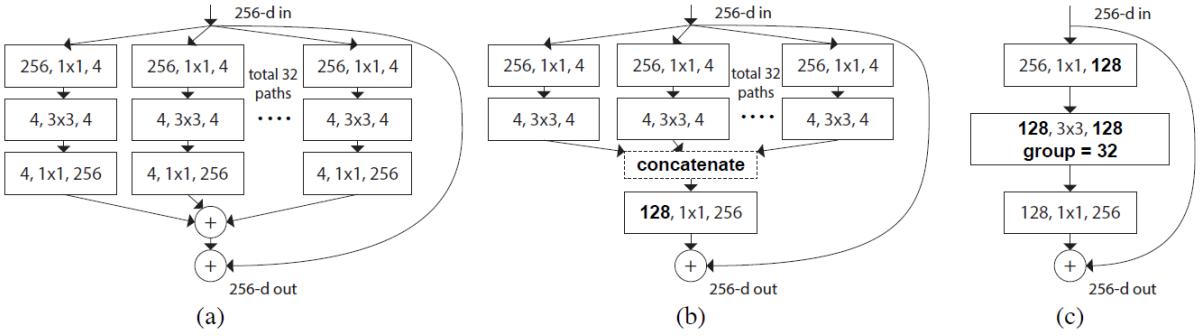


$$\mathcal{F}(\mathbf{x}) = \sum_{i=1}^{C} \mathcal{T}_i(\mathbf{x})$$

In contrast to “Network-in-Network”, it is “Network-in-Neuron” expands along a new dimension. Instead of linear function in a simple neuron that wi times xi in each path, a nonlinear function is performed for each path.

A new dimension C is introduced, called “Cardinality”. The dimension of cardinality controls the number of more complex transformations.

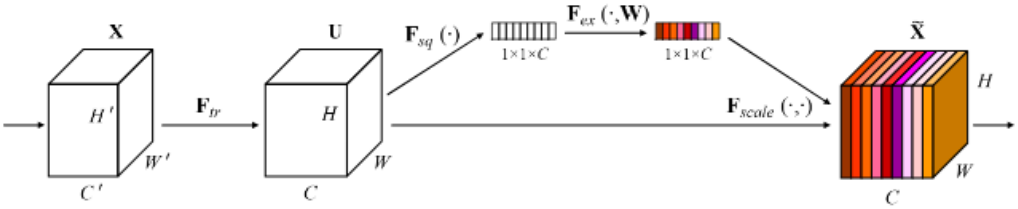
ACV-Question Bank



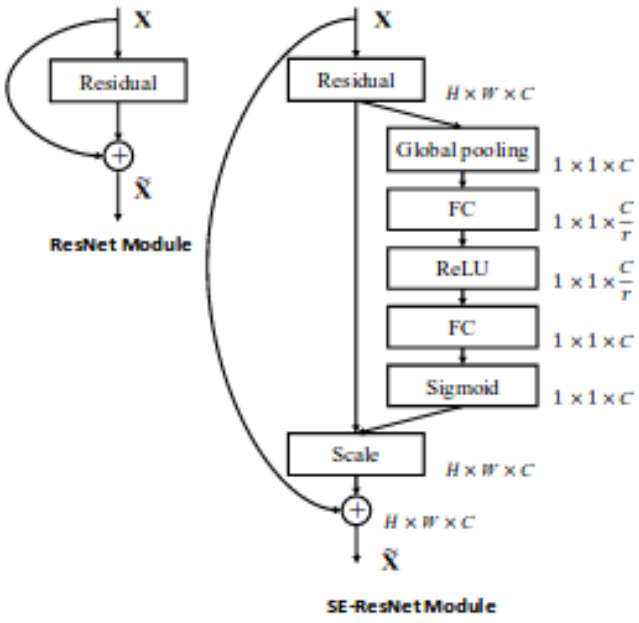
Here we see Resnext architecture in A and Inception-resnet in B and the Grouped Convolution of AlexNet in C

SE Net

One possible mistake we usually do when convolving is that we aggregate all the channels information, sum them up and forward, doing so we are really missing out lot of information on channel dependencies. To make it questionable, Are all the feature maps I have learned are really important? This paper investigated a new architectural design — the channel relationship, by introducing a new architectural unit, which we term as "Squeeze-and-excitation" (SE) block.



The SE block tries to use global information to selectively emphasize informative features and suppress less useful once. In literal terms, it tries to add weights to each and every feature map in the layer.



ACV-Question Bank

The squeeze operation, does a global max pooling channel wise. This kind of aggregates channel information and keeps it in a lower dimension.

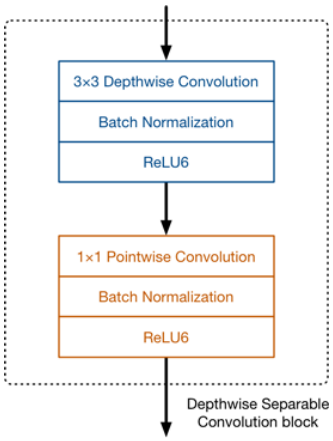
The excitation operation kinds of need to decide which of the feature maps are really important or have signal them. Learning this is done using a new layer FC layer in between with a Relu layer. In the end a sigmoid layer is applied. The sigmoid activations act as channel weights adapted to the input-specific descriptor x. SE block intrinsically introduces dynamics conditioned on the input, helping boost feature discriminability.

SE block can be used with any standard architectures. The authors have tested it on ResNet, ResNeXt, inception, inception-resnet etc. There will be very minute increments in-terms of params and computations (GFLOPS) because of extra layers like FC and pooling operations respectively.

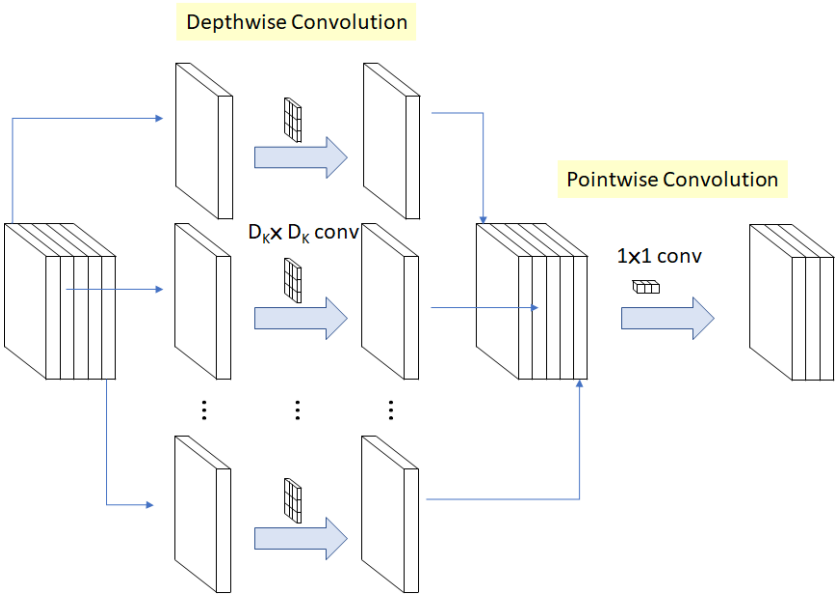
Moblie Net

The big idea behind MobileNet V1 is that convolutional layers, which are essential to computer vision tasks but are quite expensive to compute, can be replaced by so-called depthwise separable convolutions.

The job of the convolution layer is split into two subtasks: first there is a depthwise convolution layer that filters the input, followed by a 1×1 (or pointwise) convolution layer that combines these filtered values to create new features:



Depthwise separable convolution is a depthwise convolution followed by a pointwise convolution as follows:



Together, the depthwise and pointwise convolutions form a “depthwise separable” convolution block. It does approximately the same thing as traditional convolution but is much faster.

The full architecture of MobileNet V1 consists of a regular 3×3 convolution as the very first layer, followed by 13 times the above building block.

There are no pooling layers in between these depthwise separable blocks. Instead, some of the depthwise layers have a stride of 2 to reduce the spatial dimensions of the data. When that happens, the corresponding pointwise layer also doubles the number of output channels. If the input image is 224×224×3 then the output of the network is a 7×7×1024 feature map.

Full Architecture of MobileNet

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

ACV-Question Bank

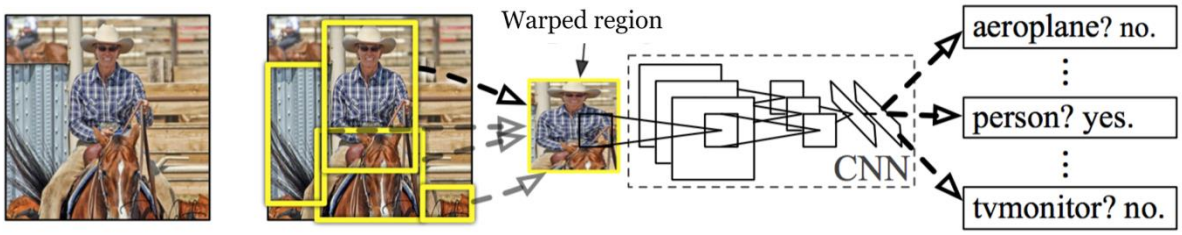
R CNN

R-CNN (Girshick et al., 2014) is short for “Region-based Convolutional Neural Networks”. The main idea is composed of two steps. First, using selective search, it identifies a manageable number of bounding-box object region candidates (“region of interest” or “RoI”). And then it extracts CNN features from each region independently for classification. How R-CNN works can be summarized as follows:

Pre-train a CNN network on image classification tasks; for example, VGG or ResNet trained on ImageNet dataset. The classification task involves N classes.

Propose category-independent regions of interest by selective search (~2k candidates per image). Those regions may contain target objects and they are of different sizes.

Region candidates are warped to have a fixed size as required by CNN.



Continue fine-tuning the CNN on warped proposal regions for K + 1 classes; The additional one class refers to the background (no object of interest). In the fine-tuning stage, we should use a much smaller learning rate and the mini-batch oversamples the positive cases because most proposed regions are just background.

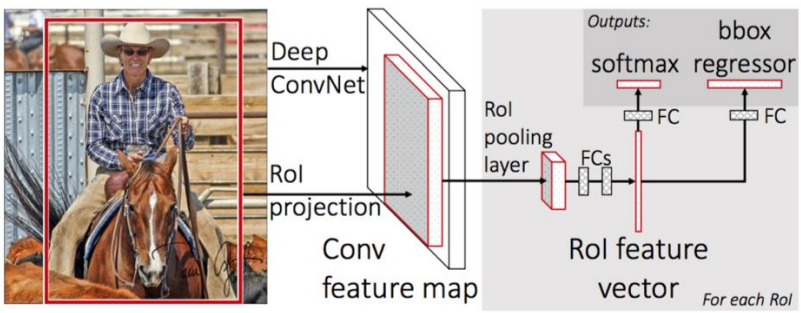
Given every image region, one forward propagation through the CNN generates a feature vector. This feature vector is then consumed by a binary SVM trained for each class independently.

The positive samples are proposed regions with IoU (intersection over union) overlap threshold >= 0.3, and negative samples are irrelevant others.

To reduce the localization errors, a regression model is trained to correct the predicted detection window on bounding box correction offset using CNN features.

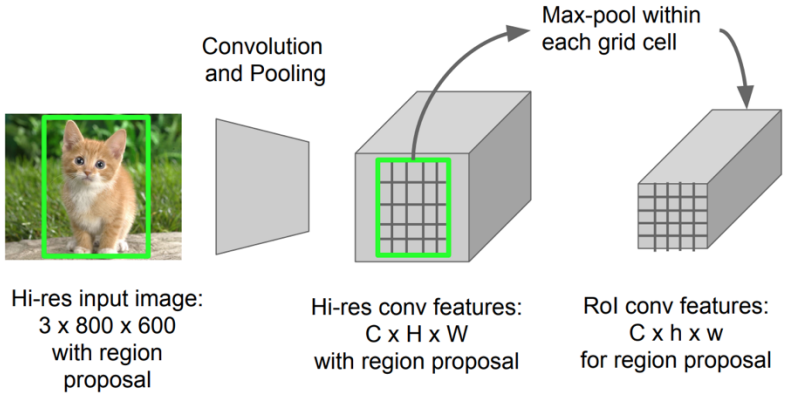
Fast R-CNN

To make R-CNN faster, Girshick (2015) improved the training procedure by unifying three independent models into one jointly trained framework and increasing shared computation results, named Fast R-CNN. Instead of extracting CNN feature vectors independently for each region proposal, this model aggregates them into one CNN forward pass over the entire image and the region proposals share this feature matrix. Then the same feature matrix is branched out to be used for learning the object classifier and the bounding-box regressor. In conclusion, computation sharing speeds up R-CNN.



ACV-Question Bank

ROI pooling is a type of max pooling to convert features in the projected region of the image of any size, h x w, into a small fixed window, H x W. The input region is divided into H x W grids, approximately every subwindow of size h/H x w/W. Then apply max-pooling in each grid.



How Fast R-CNN works is summarized as follows; many steps are same as in R-CNN:

First, pre-train a convolutional neural network on image classification tasks.

Propose regions by selective search (~2k candidates per image).

Alter the pre-trained CNN:

- Replace the last max pooling layer of the pre-trained CNN with a RoI pooling layer. The RoI pooling layer outputs fixed-length feature vectors of region proposals. Sharing the CNN computation makes a lot of sense, as many region proposals of the same images are highly overlapped.
- Replace the last fully connected layer and the last softmax layer (K classes) with a fully connected layer and softmax over K + 1 classes.

Finally the model branches into two output layers:

- A softmax estimator of K + 1 classes (same as in R-CNN, +1 is the "background" class), outputting a discrete probability distribution per RoI.
- A bounding-box regression model which predicts offsets relative to the original RoI for each of K classes.

Loss Function

The loss function sums up the cost of classification and bounding box prediction: □=□cls+□boxL=Lcls+Lbox. For "background" RoI, □boxLbox is ignored by the indicator function 𝟙[u≥1]1[u≥1], defined as:

$$\mathbb{1}[u >= 1] = \begin{cases} 1 & \text{if } u \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

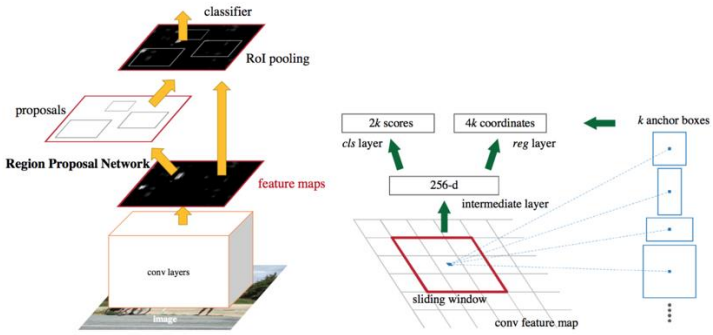
The overall loss function is:

$$\mathcal{L}(p, u, t^u, v) = \mathcal{L}_{\text{cls}}(p, u) + \mathbb{1}[u \geq 1]\mathcal{L}_{\text{box}}(t^u, v)$$
$$\mathcal{L}_{\text{cls}}(p, u) = -\log p_u$$
$$\mathcal{L}_{\text{box}}(t^u, v) = \sum_{i \in \{x,y,w,h\}} L_1^{\text{smooth}}(t_i^u - v_i)$$

ACV-Question Bank

Faster R-CNN

An intuitive speedup solution is to integrate the region proposal algorithm into the CNN model. Faster R-CNN (Ren et al., 2016) is doing exactly this: construct a single, unified model composed of RPN (region proposal network) and fast R-CNN with shared convolutional feature layers.



Working of Faster R- CNN

Pre-train a CNN network on image classification tasks.

Fine-tune the RPN (region proposal network) end-to-end for the region proposal task, which is initialized by the pre-train image classifier. Positive samples have IoU (intersection-over-union) > 0.7, while negative samples have IoU < 0.3.

- Slide a small n x n spatial window over the conv feature map of the entire image.
- At the center of each sliding window, we predict multiple regions of various scales and ratios simultaneously. An anchor is a combination of (sliding window center, scale, ratio). For example, 3 scales + 3 ratios => k=9 anchors at each sliding position.

Train a Fast R-CNN object detection model using the proposals generated by the current RPN

Then use the Fast R-CNN network to initialize RPN training. While keeping the shared convolutional layers, only fine-tune the RPN-specific layers. At this stage, RPN and the detection network have shared convolutional layers!

Finally fine-tune the unique layers of Fast R-CNN

Step 4-5 can be repeated to train RPN and Fast R-CNN alternatively if needed.

Loss Function in Faster RCNN

The multi-task loss function combines the losses of classification and bounding box regression:

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{box}}$$

$$\mathcal{L}(\{p_i\}, \{t_i\}) = \frac{1}{N_{\text{cls}}} \sum_i \mathcal{L}_{\text{cls}}(p_i, p_i^*) + \frac{\lambda}{N_{\text{box}}} \sum_i p_i^* \cdot L_1^{\text{smooth}}(t_i - t_i^*)$$

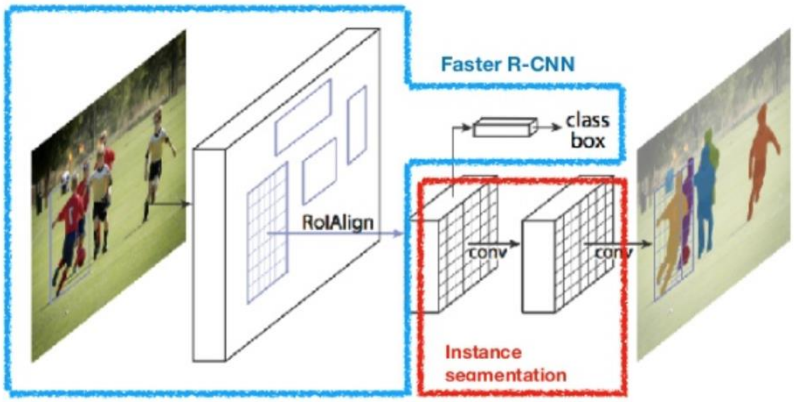
where $\mathcal{L}_{\text{cls}}$ is the log loss function over two classes, as we can easily translate a multi-class classification into a binary classification by predicting a sample being a target object versus not. L_1^{smooth} is the smooth L1 loss.

$$\mathcal{L}_{\text{cls}}(p_i, p_i^*) = -p_i^* \log p_i - (1 - p_i^*) \log(1 - p_i)$$

ACV-Question Bank

Mask RCNN

Mask R-CNN (He et al., 2017) extends Faster R-CNN to pixel-level image segmentation. The key point is to decouple the classification and the pixel-level mask prediction tasks. Based on the framework of Faster R-CNN, it added a third branch for predicting an object mask in parallel with the existing branches for classification and localization. The mask branch is a small fully-connected network applied to each RoI, predicting a segmentation mask in a pixel-to-pixel manner.



Because pixel-level segmentation requires much more fine-grained alignment than bounding boxes, mask R-CNN improves the RoI pooling layer (named “RoIAlign layer”) so that RoI can be better and more precisely mapped to the regions of the original image.

Summary of RCNN