

Final Report, SoS 2020

Neural Network and Deep Learning for Computer Vision

Purvi Poonia

Roll no: 190050087

Mentor: Bhavesh Patil

April - June 2020



Introduction

In computer science, *artificial intelligence*, sometimes called *machine intelligence*, is intelligence demonstrated by machines, in contrast to the natural intelligence displayed by humans and animals.

Machine Learning is a subset of AI which provides machines the ability to learn and automatically improve from experience without being explicitly programmed.

Neural Networks are at the heart of any supervised machine learning task.

Computer Vision is the field that deals how computers can gain high level understanding from digital images or videos. From engineering perspective, it seeks to understand and automate tasks that human visual system can do.

Computer Vision has become ubiquitous in our society, with applications in search, image understanding, apps, mapping, medicine, drones, and self-driving cars. Core to many of these applications are visual recognition tasks such as image classification, localization and detection. Recent developments in neural network (aka “deep learning”) approaches have greatly advanced the performance of these state-of-the-art visual recognition systems.

This report talks about the basic concepts which are the heart of any Computer Vision application.

Acknowledgement

This report was written as a part of Summer of Science-2020 organised by the Maths and Physics Club, IIT Bombay. The content here is highly inspired by the CS231n Convolutional Neural Network for Visual Recognition and Coursera course Convolutional Neural Networks by Andrew NG and at times might simply be paraphrasing of the course content.

I would like to thank my mentor Bhavesh Patil who was there to solve even the smallest doubts and guided me to the right resources. Without his help it would not have been possible to develop a deep understanding about the topics.

Also, I thank Maths and Physics Club, IIT Bombay for the efforts they have put to organize Summer of Science,2020.

Contents

1. Image Classification

- 1.1 What is Image Classification
- 1.2 Challenges
- 1.3 Image Classification Pipeline

2. Nearest Neighbour Classifiers

- 2.1 k-Nearest Neighbour Classifier
- 2.2 Validation sets for hyperparameter tuning
- 2.3 Disadvantages

3. Neural Networks – Introduction

4. Activation Function

- 4.1 Sigmoid
- 4.2 Tanh
- 4.3 ReLU

5. Loss function

- 5.1 Multiclass Support Vector Machine
- 5.2 Softmax Classifier

6. Gradient Descent

- 6.1 Vanilla Update
- 6.2 Momentum Update
- 6.3 Adagrad
- 6.4 RMS Prop
- 6.5 Adam
- 6.6 Visualizing gradient descent

7. Backpropagation

8. Overfitting and Regularization

- 8.1 The problem of Overfitting

- 8.2 Data Augmentation
- 8.3 Regularization Techniques

9. Types of Neural Nets

- 9.1 Regular Neural Networks
- 9.2 Convolutional Neural Networks
 - 9.2.1 Input
 - 9.2.2 Convolutional Layer
 - 9.2.3 Activation Layer
 - 9.2.4 Pooling
 - 9.2.5 Fully-connected layer

10. Data Preprocessing

- 10.1 Mean Subtraction
- 10.2 Normalization

11. Training Neural Networks

- 11.1 Layer Patterns

12. Neural Style Transfer

- 12.1 What is Neural Style Transfer
- 12.2 Cost Function

13. Localization and Detection

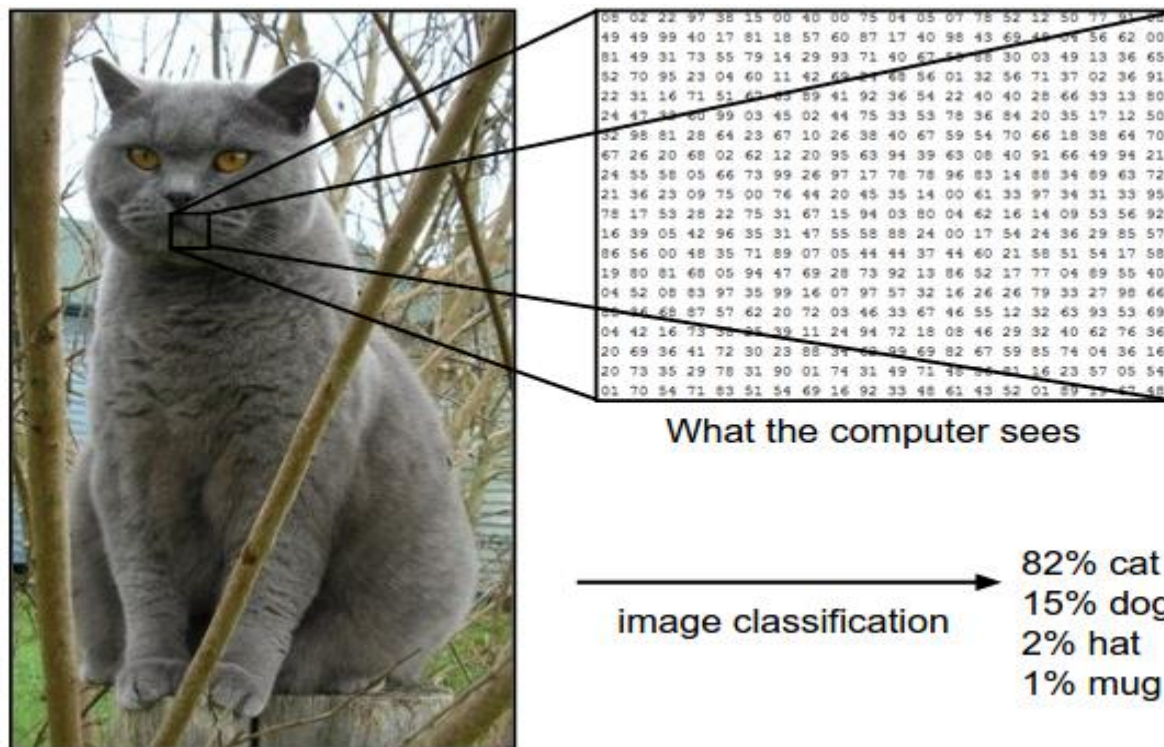
- 13.1 Localization
- 13.2 Detection
 - 13.2.1 Sliding Windows Detection Algorithm
 - 13.2.2 Convolutional Implementation of Sliding Windows Algorithm
 - 13.2.3 YOLO Algorithm – You Only Look Once

1.IMAGE CLASSIFICATION

~A core task in computer vision

1.1 What is Image Classification?

Image classification refers to a process in computer vision that can classify an image according to its visual content. We assume a set of discrete class-labels for example cat, dog, truck, etc. and then pass the input image through the classification algorithm which outputs the class label (example cat) or the probability that the input is a particular class (“there’s a 90% probability that this input is a cat”).



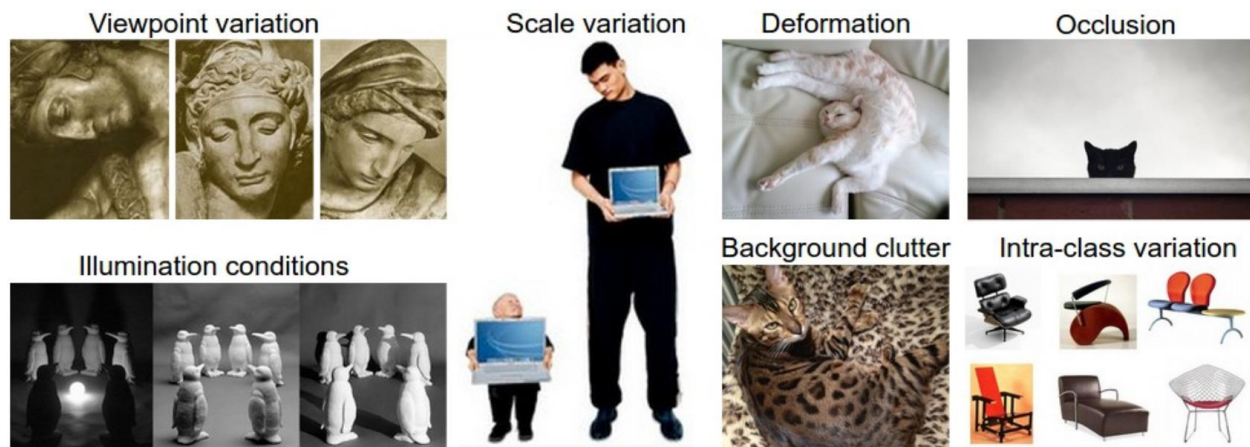
The raw representation of images is a 3-D array of numbers, with integers between [0, 255] which represent the brightness values.

1.2 Challenges

Task of recognizing a visual concept (e.g. cat) is relatively trivial for a human to perform, but it is worth considering the challenges involved from the perspective of a Computer Vision algorithm.

- **Viewpoint variation:** A single instance of an object can be oriented in many ways with respect to the camera.

- **Scale variation:** Visual classes often exhibit variation in their size (size in the real world, not only in terms of their extent in the image).
- **Deformation:** Many objects of interest are not rigid bodies and can be deformed in extreme ways.
- **Occlusion:** The objects of interest can be occluded. Sometimes only a small portion of an object (as little as few pixels) could be visible.
- **Illumination conditions:** The effects of illumination are drastic on the pixel level.
- **Background clutter:** The objects of interest may blend into their environment, making them hard to identify.
- **Intra-class variation:** The classes of interest can often be relatively broad, such as chair. There are many different types of these objects, each with their own appearance.



1.3 The image classification pipeline

We've seen that the task in Image Classification is to take an array of pixels that represents a single image and assign a label to it. Our complete pipeline can be formalized as follows:

- **Input:** Our input consists of a set of N images, each labeled with one of K different classes. We refer to this data as the training set.
- **Learning:** Our task is to use the training set to learn what every one of the classes looks like. We refer to this step as training a classifier, or learning a model.

- **Evaluation:** In the end, we evaluate the quality of the classifier by asking it to predict labels for a new set of images (test set) that it has never seen before. We will then compare the true labels of these images to the ones predicted by the classifier. Intuitively, we're hoping that a lot of the predictions match up with the true answers (which we call the ground truth).

2. Nearest Neighbor Classifier

The Nearest Neighbor Classifier is a basic approach to an image classification task. This classifier has nothing to do with Convolutional Neural Networks and it is very rarely used in practice, but it will allow us to get an idea about the basic approach to an image classification problem.

The nearest neighbor classifier will take a test image, compare it to every single one of the training images, and predict the label of the closest training image.

Measure of distance:

1. L1 (Manhattan) Distance :

Simplest possibilities is to compare the images pixel by pixel and add up all the differences. The images are represented as vectors

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

| test image | | | | | training image | | | | | pixel-wise absolute value differences | | | | |
|------------|----|-----|-----|---|----------------|----|-----|-----|---|---------------------------------------|----|----|-----|-------|
| 56 | 32 | 10 | 18 | | 10 | 20 | 24 | 17 | | 46 | 12 | 14 | 1 | |
| 90 | 23 | 128 | 133 | | 8 | 10 | 89 | 100 | | 82 | 13 | 39 | 33 | |
| 24 | 26 | 178 | 200 | - | 12 | 16 | 178 | 170 | = | 12 | 10 | 0 | 30 | |
| 2 | 0 | 255 | 220 | | 4 | 32 | 233 | 112 | | 2 | 32 | 22 | 108 | → 456 |

2. L2 (Euclidean) Distance:

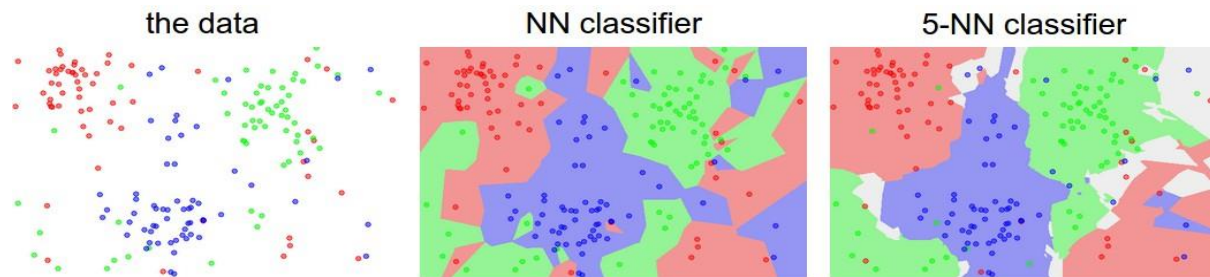
Here we compute the pixel wise difference, square all of them, add them up and finally take the square root.

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

2.1 k - Nearest Neighbor Classifier

Instead of using just one nearest example we can do better by using a **k-Nearest Neighbor Classifier**. Instead of finding the single closest image in the training set, we will find the top k closest images, and have them vote on the label of the test image. In particular, when $k = 1$, we recover the Nearest Neighbor classifier.

Intuitively, higher values of k have a smoothing effect that makes the classifier more resistant to outliers. k is a hyperparameter.



In the case of a NN classifier, outlier datapoints (e.g. green point in the middle of a cloud of blue points) create small islands of likely incorrect predictions, while the 5-NN classifier smooths over these irregularities. Also the gray regions in the 5-NN image are caused by

2.2 Validation sets for Hyperparameter tuning

The k -nearest neighbor classifier requires a setting for k . But what number works best? Additionally, there are many different distance functions we could have used: L1 norm, L2 norm, there are many other choices weren't even considered (e.g. dot products). These choices are called hyperparameters and they come up very often in the design of many Machine Learning algorithms that learn from data. It's often not obvious what values/settings one should choose. These hyperparameters can be tuned by splitting our training set in two: a slightly smaller training set, and what we call a validation set. This validation set is essentially used as a fake test set to tune the hyper-parameters. In cases where the size of training data (and therefore also the validation data) is small, people sometimes use a more sophisticated technique for hyperparameter tuning called cross-validation.

2.3 Disadvantages

- The images that are nearby each other are much more a function of the general color distribution of the images, or the type of background rather than their semantic identity. For example, a horse can be seen very near a truck because of similar background.



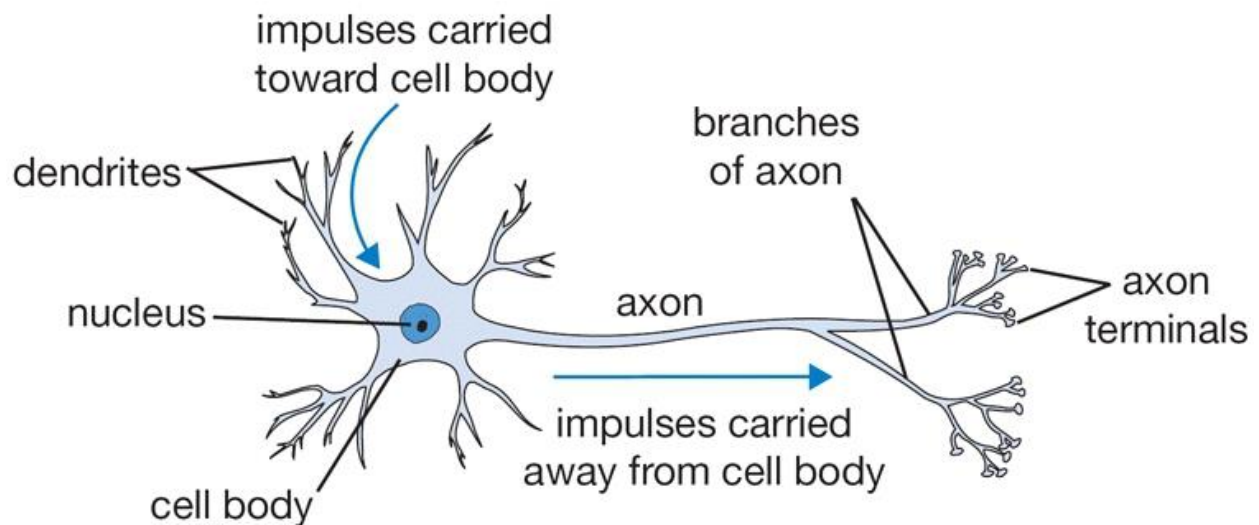
- The classifier must remember all of the training data and store it for future comparisons with the test data. This is space inefficient because datasets may easily be gigabytes in size.
- Classifying a test image is expensive since it requires a comparison to all training images.

3. Neural Networks -Introduction

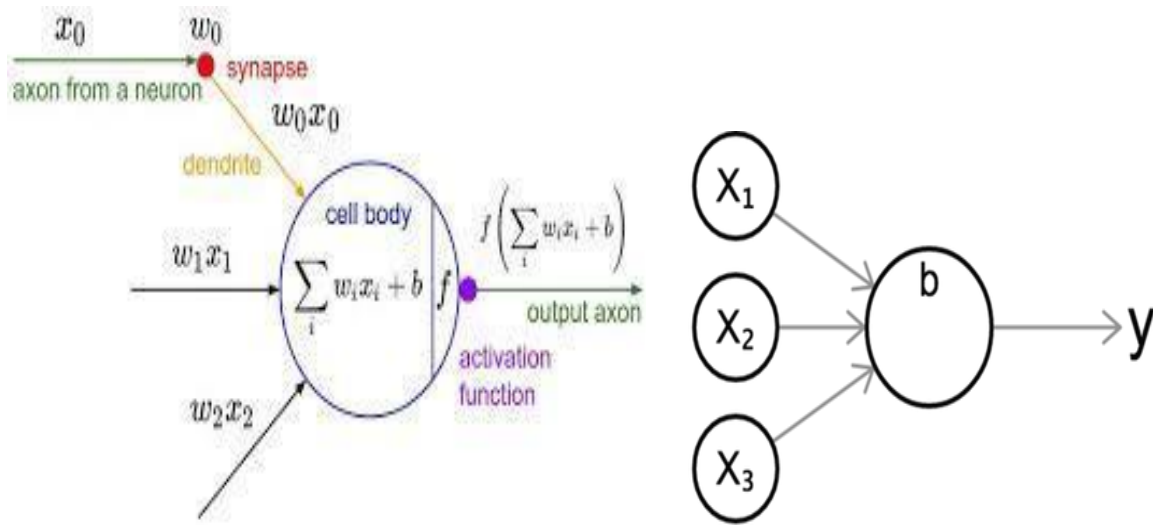
The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems. The basic computational unit of the brain is a **neuron**. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately $10^{14} - 10^{15}$ **synapses**. Each neuron receives input signals from its **dendrites** and produces output signals along its (single) **axon**. The axon eventually branches out and connects via synapses to dendrites of other neurons.

In the computational model of a neuron, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. w_0x_0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitatory (positive weight) or inhibitory (negative weight)) of one neuron on another.

In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can *fire*, sending a spike along its axon.



Since neural Networks are inspired by human brain, just like human brains the basic building block is called a Neuron. Its functionality is similar to a human neuron, i.e. it takes in some inputs and fires an output. In purely mathematical terms, a neuron in the machine learning world is a placeholder for a mathematical function, and its only job is to provide an output by applying the function on the inputs provided.



Artificial Neuron

As shown above each input as a weight (w) associated with it and each neuron has a bias (b) term.

4. Activation Functions

The function used in a neuron is generally termed as an activation function.

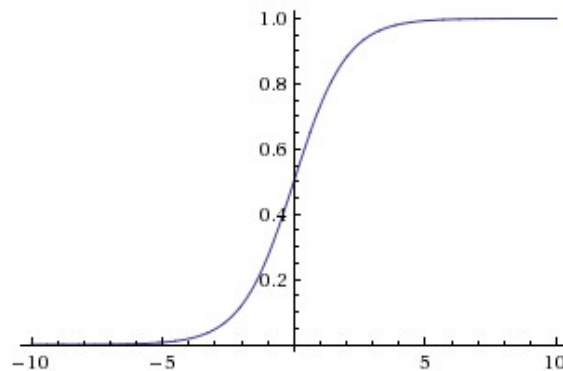
Commonly used Activation Functions:

4.1 Sigmoid

The sigmoid non-linearity has the mathematical form

$$\sigma(x) = \frac{1}{(1 + e^{-x})}$$

It takes a real-valued number and “squashes” it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1).



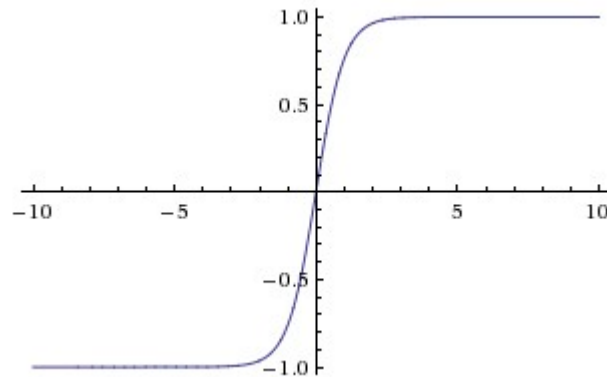
Major drawbacks:

- Sigmoids saturate and kill gradients. When the neuron’s activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. During backpropagation, this (local) gradient will be multiplied to the gradient of this gate’s output for the whole objective. Therefore, if the local gradient is very small, it will effectively “kill” the gradient and almost no signal will flow through the neuron to its weights and recursively to its data.
- Sigmoid outputs are not zero-centered. This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = w^T x + b$), then the gradient on the weights w will during backpropagation become either all be positive, or all

negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights.

4.2 Tanh

The tanh non-linearity squashes a real-valued number to the range $[-1, 1]$. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x) = 2\sigma(2x) - 1$.

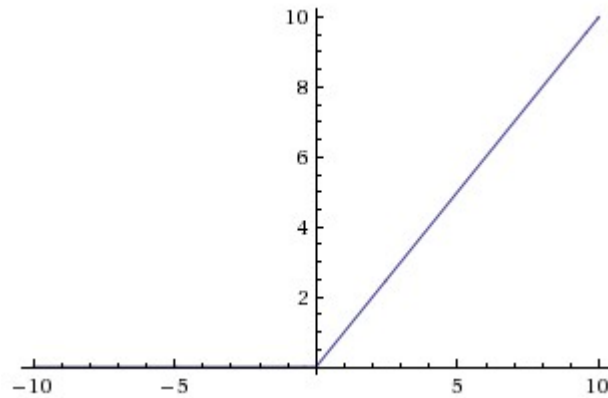


4.3 ReLU

The Rectified Linear Unit has mathematical form

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

The activation is simply thresholded at zero .



There are several pros and cons to using the ReLUs:

- It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions
- Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- Unfortunately, ReLU units can be fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. With a proper setting of the learning rate this is less frequently an issue.

5. Loss function

We don't have control over the data (x_i, y_i) (it is fixed and given), but we do have control over weights and we want to set them so that the predicted class scores are consistent with the ground truth labels in the training data.

Loss function is a way of quantifying how bad our current set of weights is.

Intuitively, the loss will be high if we're doing a poor job of classifying the training data, and it will be low if we're doing well.

5.1 Multiclass Support Vector Machine (SVM)

SVM is set up such that it wants the score of the correct class to be higher than all other class scores by at least a margin of Δ .

We are given the pixels of image (x_i) and the label (y_i) that specifies the index of the correct class. The score function takes the pixels and computes the vector $f(x_i, W)$ of class scores, which we will abbreviate to s . For example, the score for the j -th class is the j -th element: $s_j = f(x_i, W)_j$. The Multiclass SVM loss for the i -th example is then formalized as follows:

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

5.2 Softmax classifier

In the Softmax classifier, the function mapping $f(x_i, W) = Wx_i$ stays unchanged, but we now interpret these scores as the unnormalized log probabilities for each class.

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \quad \text{or equivalently} \quad L_i = -f_{y_i} + \log \sum_j e^{f_j}$$

Where, $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$ is known as the softmax function. It takes a vector of arbitrary real-valued scores (in z) and squashes it to a vector of values between zero and one that sum to one.

Total loss is sum over all training examples.

Optimization

The process of finding the parameters W, b which minimize the loss function is known as Optimization.

6.Gradient Descent

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**.

The parameter update looks like:

$$x' = x - lr * dx$$

In our case the function is the loss function and parameters are weights (W) and biases (b).

There are several forms of parameter updates like

6.1 Vanilla update. The simplest form of update is to change the parameters along the negative gradient direction (since the gradient indicates the direction of increase, but we usually wish to minimize a loss function). Assuming a vector of parameters x and the gradient dx , the simplest update has the form:

Vanilla update

`x += - learning_rate * dx`

The `learning_rate` is a hyperparameter - a fixed constant. When evaluated on the full dataset, and when the learning rate is low enough, this is guaranteed to make non-negative progress on the loss function.

6.2 Momentum update is another approach that almost always enjoys better converge rates on deep networks. This update can be motivated from a physical perspective of the optimization problem. In particular, the loss can be interpreted as the height of a hilly terrain (and therefore also to the potential energy since $U=mgh$ and therefore $U \propto h$). Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location. The optimization process can then be seen as equivalent to the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape.

`v = mu * v - learning_rate * dx` *# integrate velocity*

`x += v` *# integrate position*

Variable v is initialized at zero. The hyperparameter (μ) damps the velocity and reduces the kinetic energy of the system, or otherwise the particle would never come to a stop at the bottom of a hill. This parameter is usually set to values such as [0.5, 0.9, 0.95, 0.99].

6.3 Adagrad is an adaptive learning rate method.

Assume the gradient dx and parameter vector x

```
cache += dx**2
```

```
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

The variable `cache` has size equal to the size of the gradient, and keeps track of per-parameter sum of squared gradients. This is then used to normalize the parameter update step, element-wise. The weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased. The smoothing term `eps` (usually set somewhere in range from $1e-4$ to $1e-8$) avoids division by zero. A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

6.4 RMSprop is a very effective adaptive learning rate method. The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead, giving:

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
```

```
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

Here, `decay_rate` is a hyperparameter and typical values are $[0.9, 0.99, 0.999]$. The `cache` variable is “leaky”. Hence, RMSProp still modulates the learning rate of each weight based on the magnitudes of its gradients, which has a beneficial equalizing effect, but unlike Adagrad the updates do not get monotonically smaller.

6.5 Adam is a recently proposed update that looks a bit like RMSProp with momentum. The (simplified) update looks as follows:

```
m = beta1*m + (1-beta1)*dx
```

```
v = beta2*v + (1-beta2)*(dx**2)
```

```
x += - learning_rate * m / (np.sqrt(v) + eps)
```

Recommended values are $\text{eps} = 1\text{e-}8$, $\text{beta1} = 0.9$, $\text{beta2} = 0.999$. In practice Adam is currently recommended as the default algorithm to use.

The full Adam update also includes a bias correction mechanism, which compensates for the fact that in the first few time steps the vectors m, v are both initialized and therefore biased at zero, before they fully “warm up”. With the bias correction mechanism, the update looks as follows:

t is the iteration counter going from 1 to infinity

```
m = beta1*m + (1-beta1)*dx
```

```
mt = m / (1-beta1**t)
```

```
v = beta2*v + (1-beta2)*(dx**2)
```

```
vt = v / (1-beta2**t)
```

```
x += - learning_rate * mt / (np.sqrt(vt) + eps)
```

The update is now a function of the iteration as well as the other parameters.

6.5 Visualizing gradient descent

Although our parameter vector space (W, b) is high dimensional, we can gain intuition about what gradient descent is doing by considering parameter vector to be two dimensional.

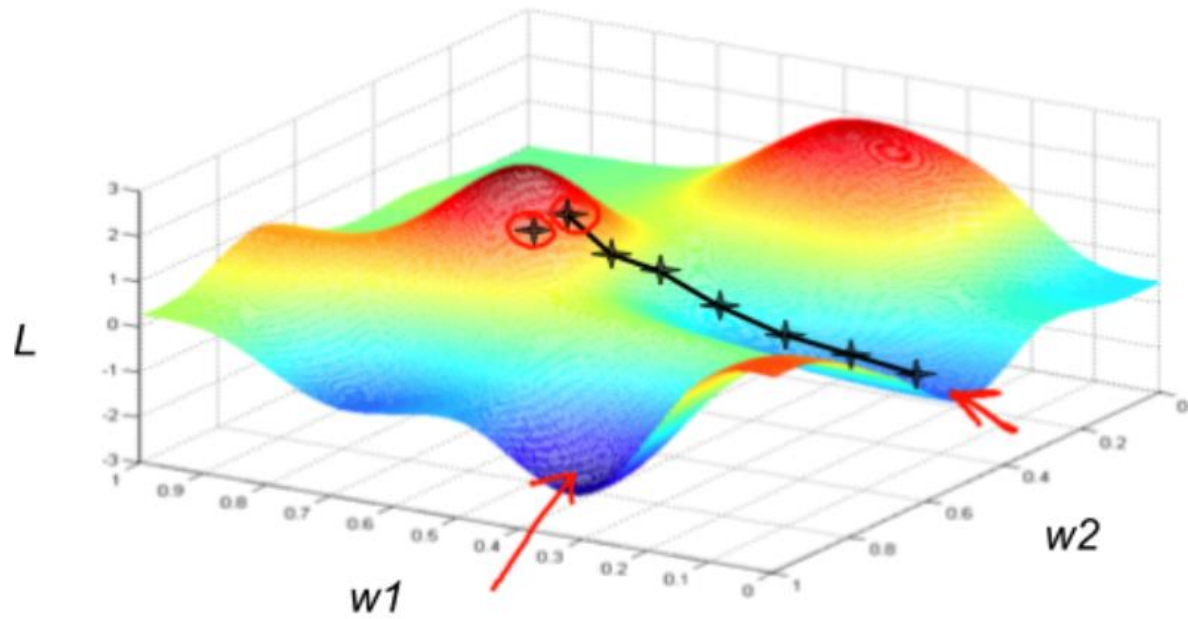
The graph below has loss function on the z-axis and the parameters on the x and y axes.

We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α , which is called the learning rate.

For example, the distance between each 'star' in the graph above represents a step determined by our parameter α . A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of L with respect to w_1 and w_2 . Depending on where one starts on the graph,

one could end up at different points. The image above shows us two different starting points that end up in two different places.



7. Backpropagation

Backpropagation is a way of computing gradients of expressions through recursive application of **chain rule**.

Given some function $f(x)$ where x is a vector of inputs and we are interested in computing the gradient of f at x (i.e. $\nabla f(x)$.)

In the specific case of neural networks, f will correspond to the loss function (L) and the inputs x will consist of the training data and the neural network weights (W) and biases (b).

Note that (as is usually the case in Machine Learning) we think of the training data as given and fixed, and of the weights as variables we have control over. Hence, even though we can easily use backpropagation to compute the gradient on the input examples x_i in practice we usually only compute the gradient for the parameters (e.g. W , b) so that we can use it to perform a parameter update.

An example for calculating the gradients with computational graphs:

This expression can be broken down into two expressions:

$q = x + y$ and $f = qz$. Moreover, we know how to compute the derivatives of both expressions separately, as seen in the previous section. f is just multiplication of q and z , so $\frac{\partial f}{\partial q} =$

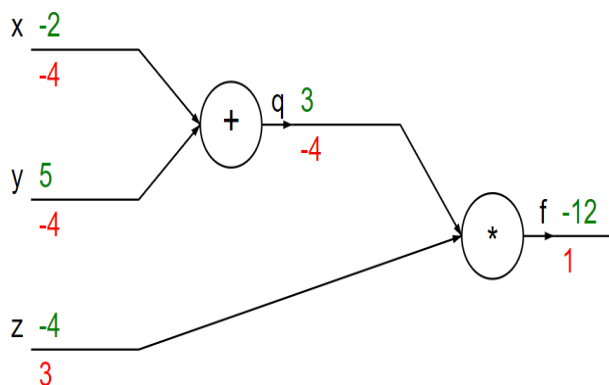
z , $\frac{\partial f}{\partial z} = q$ and q is addition of x and y so $\frac{\partial f}{\partial x} = 1$ $\frac{\partial f}{\partial y} = 1$. However, we don't necessarily

care about the gradient on the intermediate value q - the value of

$\frac{\partial f}{\partial q}$ is not useful. Instead, we are ultimately interested in the gradient of f with respect to

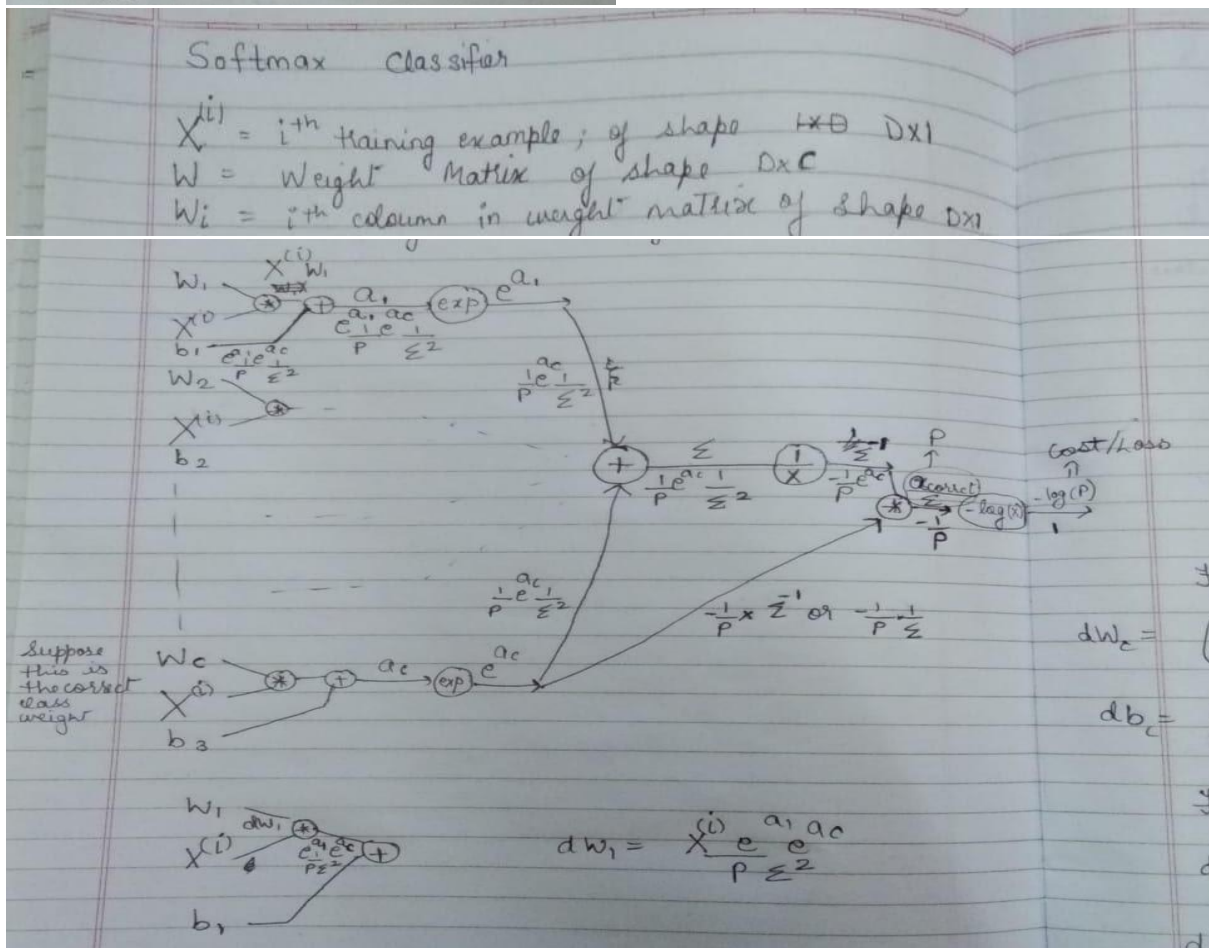
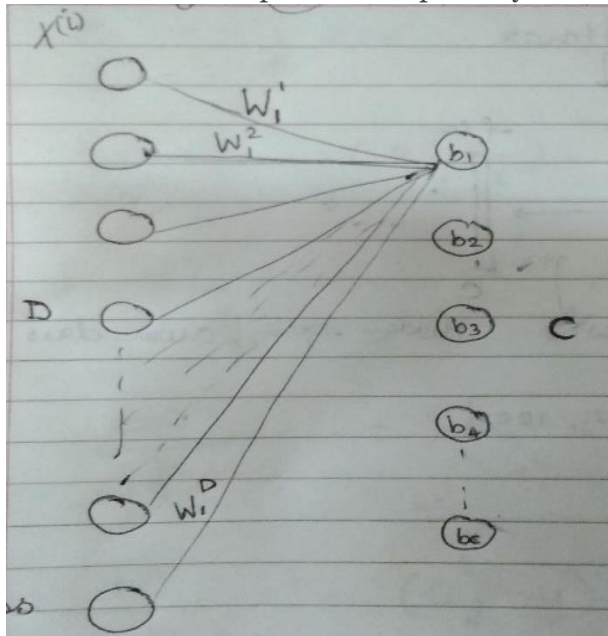
its inputs x, y, z . The chain rule tells us that the correct way to "chain" these gradient expressions together is through multiplication. For example,

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$. In practice this is simply a multiplication of the two numbers that hold the two gradients.



The **forward pass** computes values from inputs to output (shown in green). The **backward pass** then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the way to the inputs of the circuit. The gradients can be thought of as flowing backwards through the circuit.

A worked-out example for a simple 1-layer neural network (input connected to output layer)



for correct class weight -

$$dw_c = \left(\frac{1}{P} \frac{e^{a_c}}{\sum} - \frac{1}{P} \frac{1}{\sum} \right) e^{a_c} X^{(i)} = \left(\frac{e^{a_c}}{\sum} - 1 \right) X^{(i)} \quad \left(\because P = \frac{e^{a_c}}{\sum} \right)$$

$$db_c = \left(\frac{1}{P} \frac{e^{a_c}}{\sum} - \frac{1}{P} \frac{1}{\sum} \right) e^{a_c} = \left(\frac{e^{a_c}}{\sum} - 1 \right)$$

for rest -

$$dw_j = \left(\frac{1}{P} \frac{e^{a_c}}{\sum} \right) e^{a_j} X^{(i)} = \frac{e^{a_j}}{\sum} X^{(i)}$$

$$db_j = \frac{1}{P} \frac{e^{a_c}}{\sum} e^{a_j} = \frac{e^{a_j}}{\sum}$$

Where correct class scores refer to the set of weights connecting correct class neuron in final layer and all neurons in previous layer.

Backpropagation is a beautifully local process. Every gate in a circuit diagram gets some inputs and can right away compute two things: 1. its output value and 2. the local gradient of its output with respect to its inputs. Notice that the gates can do this completely independently without being aware of any of the details of the full circuit that they are embedded in. However, once the forward pass is over, during backpropagation the gate will eventually learn about the gradient of its output value on the final output of the entire circuit. Chain rule says that the gate should take that gradient and multiply it into every gradient it normally computes for all of its inputs.

8. Overfitting and Regularization

8.1 The problem of Overfitting

Models with a large number of free parameters can describe an amazingly wide range of phenomena. But even if such a model agrees well with the available data, that doesn't make it a good model. It may just mean there's enough freedom in the model that it can describe almost any data set of the given size, without capturing any genuine insights into the underlying phenomenon. When that happens, the model will work well for the existing data, but will fail to generalize to new situations. The true test of a model is its ability to make predictions in situations it hasn't been exposed to before. The lack to generalize a model is mainly because of a problem called overfitting. Our Neural Nets have a large number of weights and biases and it is quite possible that they may overfit training data.

If there is a lot of difference between training and validation accuracy then our network is overfitting.

But luckily, we have techniques for reducing the effects of overfitting.

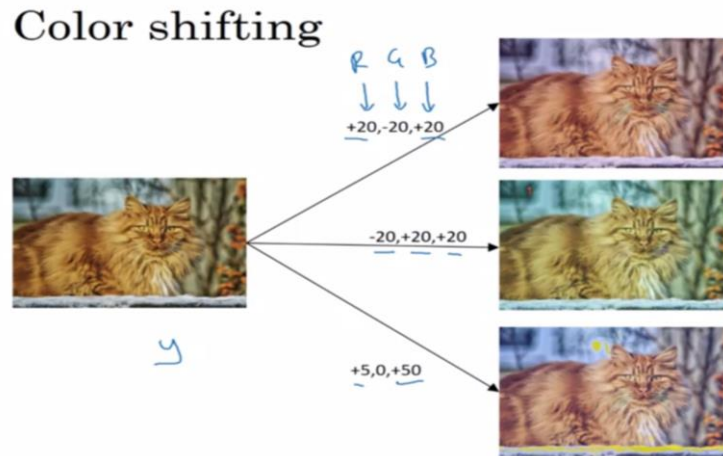
8.2 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations. This can be done by mirroring, random cropping, rotation, flipping, shearing, electric distortions, etc.



Color Shifting: The second type of data augmentation that is commonly used is color shifting. Given a picture we can add to the R, G and B channels different distortions. The motivation for this is that if maybe the sunlight was a bit yellow or maybe the in-goal illumination was a bit more yellow, that could easily change the color of an image, but

the identity of the cat or the identity of the content, the label y , just still stay the same. And so introducing these color distortions or by doing color shifting, this makes your learning algorithm more robust to changes in the colors of your images.



8.3 Regularization Techniques

- **L2**

The idea of L2 regularization is to add an extra term to the cost function, a term called the *regularization term*. It can be implemented by penalizing the squared magnitude of all parameters directly in the loss. That is, for every weight W in the network, we add the term $\frac{1}{2} * \lambda * W^2$ to the loss, where λ is the regularization strength.

- **L1**

This is similar to L2 except that instead of square of weight we add norm of the weights i.e. for each weight W we add the term $\lambda |W|$ to the loss.

Effect of L2 and L1 regularization is that we have smaller weights.

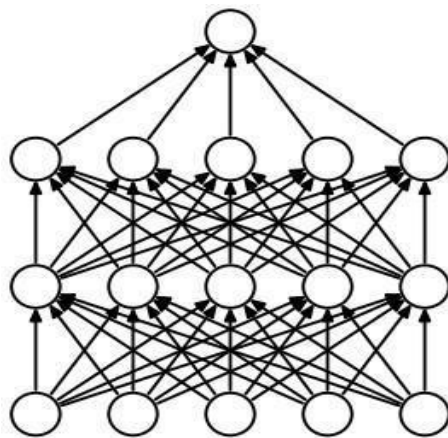
Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowed if they considerably improve the first part of the cost function. Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function. The relative importance of the two elements of the compromise depends on the value of λ . When λ is small we prefer to minimize the original cost function, but when λ is large we prefer small weights.

- **Dropout**

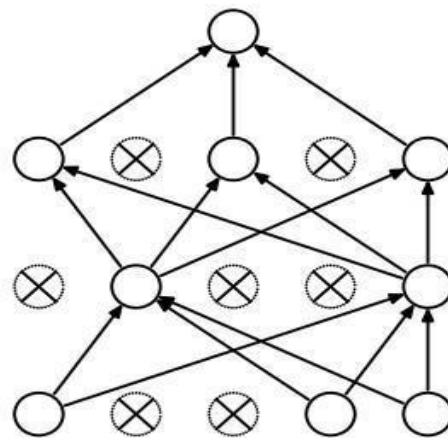
We start by randomly (and temporarily) deleting half the hidden neurons (with probability p) in the network, while leaving the input and output neurons untouched.

We forward-propagate the input x through the modified network, and then backpropagate the result, also through the modified network. After doing this over a mini-batch of examples, we update the appropriate weights and biases. We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network.

By repeating this process over and over, our network will learn a set of weights and biases.



(a) Standard Neural Net



(b) After applying dropout.

When we dropout different sets of neurons, it's rather like we're training different neural networks. And so the dropout procedure is like averaging the effects of a very large number of different networks. The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting.

8.4 Train, Test and Validation Data

We don't use all the training data for training the weights and biases; rather we split the data into training and validation data. This prevents from overfitting the training data and produce more generalized results.

The validation set is used to evaluate a given model, but this is for frequent evaluation. We use this data to fine-tune the model hyperparameters.

The test dataset provides the gold standard used to evaluate the model. It is only used once a model is completely trained (using the train and validation set).

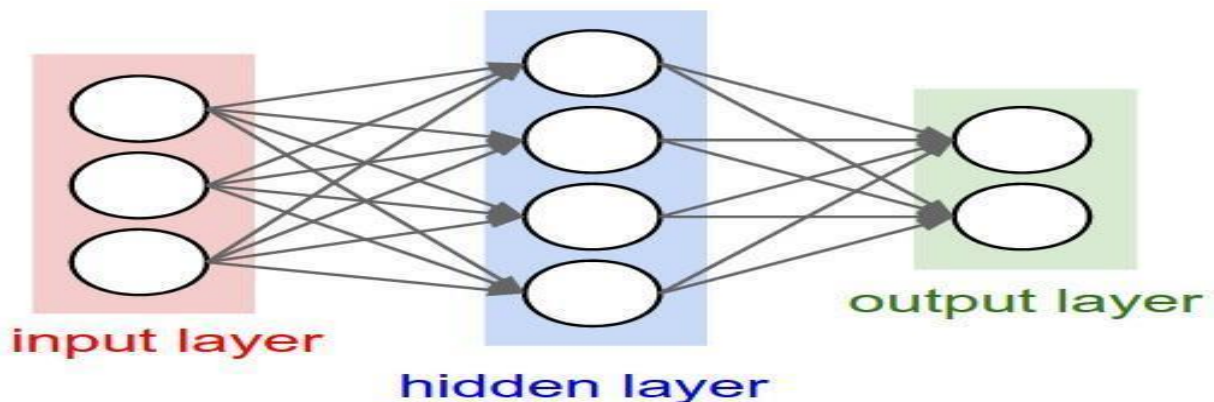
9. Types of Neural Nets

Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blob of connected neurons, Neural Network models are often organized into distinct layers of neurons.

Feed forward network: Networks where output from one layer is used as input to the next layer. There are no loops in the network, information is always fed forward.

9.1 Regular Neural Networks (Vanilla Neural Network)

The most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.



9.2 Convolutional Neural Networks (CNNs / ConvNets)

ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively).

The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input

volume. For example, a typical filter on a first layer of a ConvNet might have size $5 \times 5 \times 3$ (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume, we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

Layers used to build a ConvNet

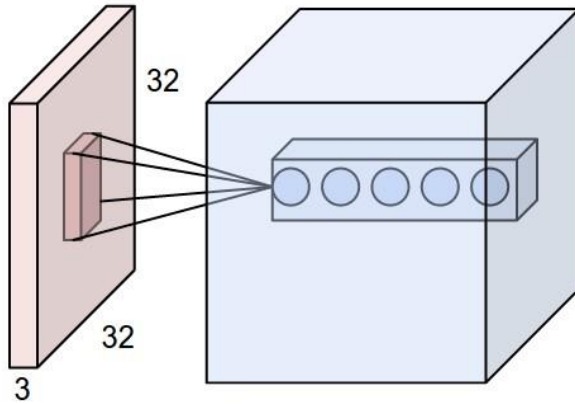
9.2.1 Input

The input to a ConvNet is usually a RGB image having three input channels.

9.2.2 Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.

Overview and intuition without brain stuff. The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size $5 \times 5 \times 3$ (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.



An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). There are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below.

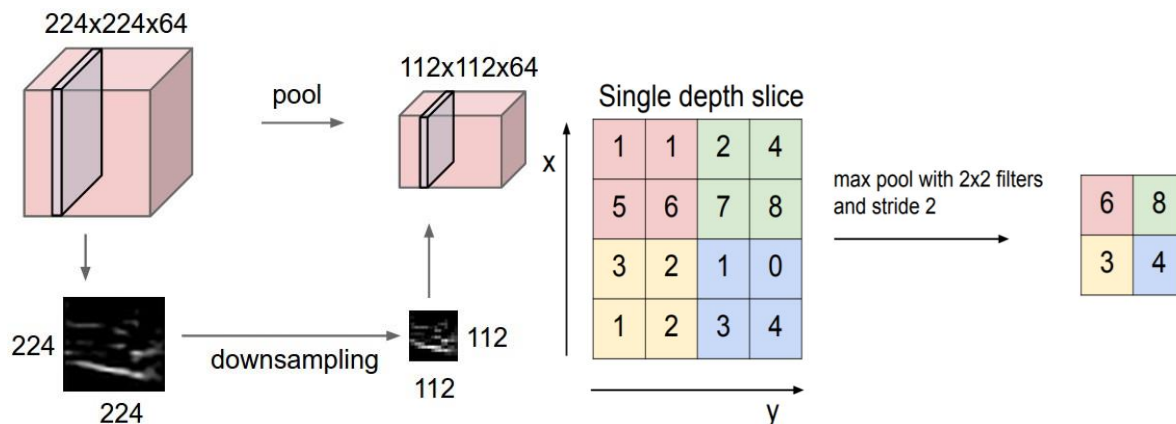
9.2.3 Activation function

One of the standard activation function is used.

9.2.4 Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially.

Max Pooling is the most commonly used pooling type.



In addition to max pooling, the pooling units can also perform other functions, such as average pooling or even L2-norm pooling. Average pooling was often used historically

but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.

9.2.5 Fully-connected layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. See the Neural Network section of the notes for more information.

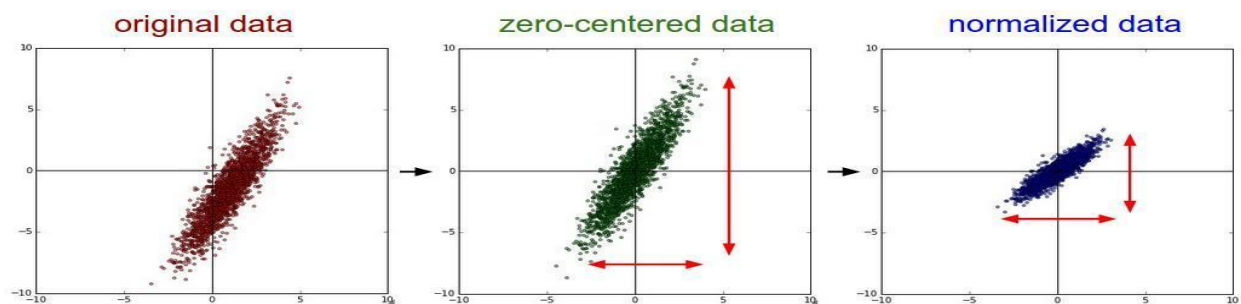
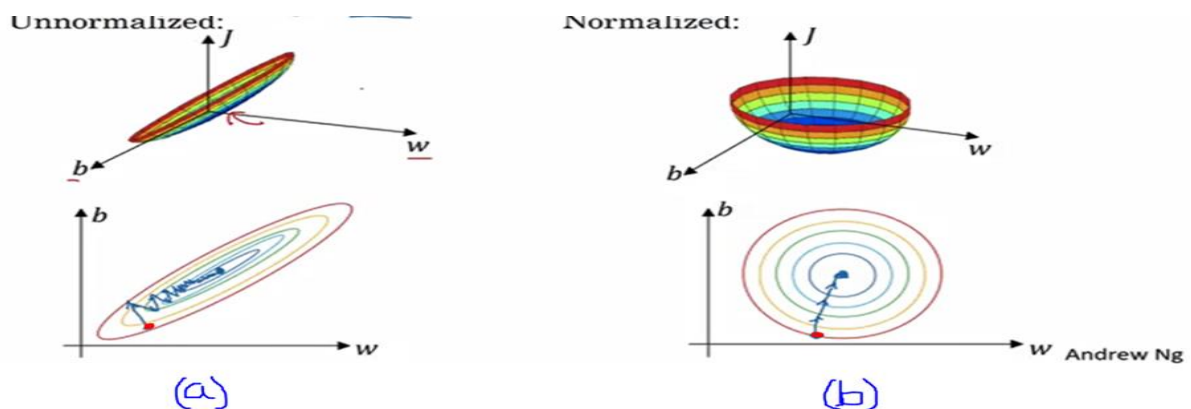
It is worth noting that the only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical.

10. Data Preprocessing

10.1 Mean subtraction is the most common form of preprocessing. It involves subtracting the mean across every individual *feature* in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension.

10.2 Normalization refers to normalizing the data dimensions so that they are of approximately the same scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered. Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively.

Normalizing the input data helps speed up training process. If the input features are spread across different scales the cost function has an elongated bowl-like shape and hence training becomes slower; whereas after normalizing cost function becomes more round.



Note:

An important point to make about the preprocessing is that any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation / test data. E.g. computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a

mistake. Instead, the mean must be computed only over the training data and then subtracted equally from all splits (train/val/test).

11. Training Neural Networks

11.1 Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows the pattern:

INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
where the * indicates repetition, and the POOL? indicates an optional pooling layer. Moreover, $N \leq 3$, $M \geq 0$, $K \geq 0$ (and usually $K < 3$).

For example, here are some common ConvNet architectures you may see that follow this pattern:

- INPUT -> FC, implements a linear classifier. Here $N = M = K = 0$.
- INPUT -> CONV -> RELU -> FC
- INPUT -> [CONV -> RELU -> POOL]*2 -> FC -> RELU -> FC. Here we see that there is a single CONV layer between every POOL layer.
- INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]*2 -> FC Here we see two CONV layers stacked before every POOL layer. This is generally a good idea for larger and deeper networks, because multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation.

It is recommended to use a stack of small Conv filters rather than one large receptive field filter.

12. NEURAL STYLE TRANSFER:

12.1 What is Neural Style Transfer?

The general idea is to take two images, and produce a new image that reflects the content of one but the artistic "style" of the other.

Neural Style Transfer (NST) is one of the most fun techniques in deep learning. As seen below, it merges two images, namely: a "content" image (C) and a "style" image (S), to create a "generated" image (G).

The generated image G combines the "content" of the image C with the "style" of image S.

In the example below, the generated image is that of the Louvre museum in Paris (content image C), mixed with a painting by Claude Monet, a leader of the impressionist movement (style image S).

Style transfer relies on separating content and style of an image. The target is to create a new image containing style of style image and content of content image (base image).



Neural Style Transfer (NST) uses a previously trained convolutional network, and builds on top of that. The idea of using a network trained on a different task and applying it to a new task is called transfer learning.

In the original NST paper, VGG-19, a 19-layer version of the VGG network. This model has already been trained on the very large ImageNet database, and thus has learned to recognize a variety of low level features (at the shallower layers) and high level features (at the deeper layers).

We feed the content and style image through the pretrained neural network and extract the content and style information respectively from them. We also feed the generated

image (which is initialized randomly) through the net to extract the content and style information from it. Using these informations and we define the loss function and then optimize it so that the generated image has the content of the content image and style of the generated image.

12.2 Cost Function

The cost function of comprises of two terms:

- **Content Cost**
- **Style Cost**

$$\begin{array}{c}
 \text{total cost function} \qquad \qquad \qquad \text{similarity of style image} \\
 \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{to generated image} \\
 \underbrace{\hspace{1.5cm}} \qquad \qquad \qquad \underbrace{\hspace{1.5cm}} \\
 J_{total}(G) = \alpha \times J_{content}(C, G) + \beta \times J_{style}(S, G) \\
 \underbrace{\hspace{1.5cm}} \qquad \qquad \qquad \underbrace{\hspace{1.5cm}} \\
 \text{similarity of content} \qquad \qquad \qquad \text{similarity of style image} \\
 \text{image to generated image} \qquad \qquad \qquad \text{to generated image}
 \end{array}$$

Where α and β are the weighting factors for content and style reconstruction, respectively

12.2.1 Content Cost

It measures how similar is the content of the original content image and the generated image.

We chose a layer (somewhere in the middle of the network) and use the activations of that layer (l) to calculate the content cost. If l is very small then the generated image pixel values will be very similar to the content image while choosing a large l then it will be like suppose there is a dog in the content image then there will be a dog somewhere in the generated image. So choosing l neither too shallow nor too deep works well.

If we choose layer l then,

$$J_{content}(C, G) = \frac{1}{2} \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2$$

$a^{(C)}, a^{(G)}$ are activations of layer l of content and generated images respectively.

And so, if we use the degree of correlation between channels as a measure of the style, then what you can do is measure the degree to which in your generated image, this first channel is correlated or uncorrelated with the second channel and that will tell you in the generated image how often this type of vertical texture occurs or doesn't occur with this orange-ish tint and this gives you a measure of how similar is the style of the generated image to the style of the input style image.

12.2.2 Style Cost:

The style cost is defined as

$$J_{style}^{[l]}(S, G) = \sum_k \sum_{k'} \left(G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)} \right)^2$$

Here we use layer l 's activations as a measure of “style”. Style is defined as correlation between activations across channels.

G is the style (or gram) matrix which is used as a measure of style. The kk' element of the matrix is given by

$$G_{kk'}^{[l](G)} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a_{i,j,k}^{[l](G)} a_{i,j,k'}^{[l](G)}.$$

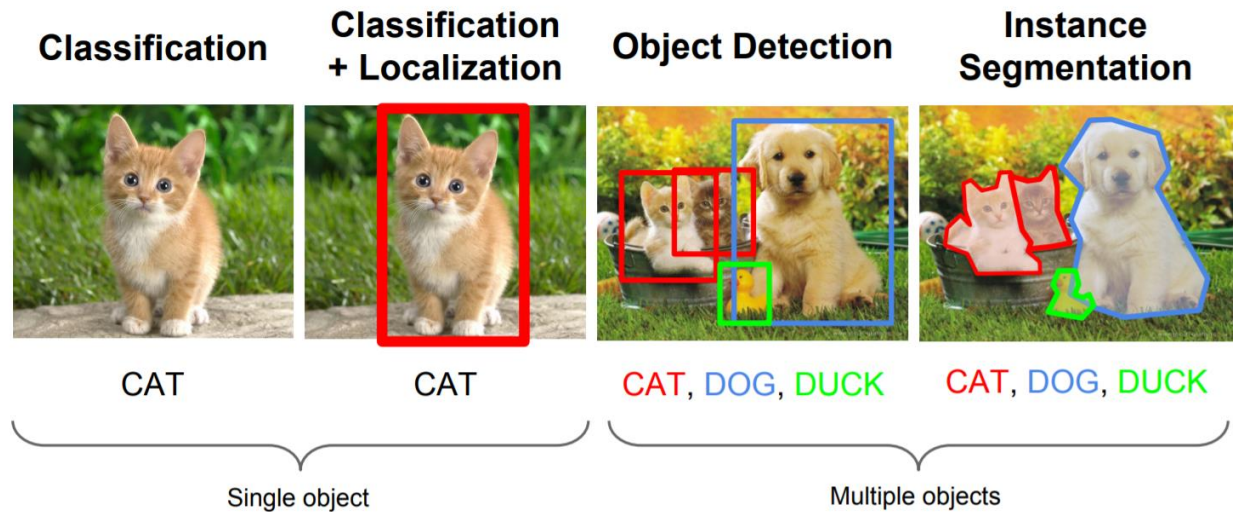
And this is a measure of how correlated are the layers k and k' . Where the superscript (G) refers to the generated image.

The final style cost is usually calculated using multiple layers both earlier and later.

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S, G)$$

The loss function is optimized using any standard optimizer to perform gradient descent to change the generated image such that it decreases the loss after each iteration.

13. Localization and Detection:

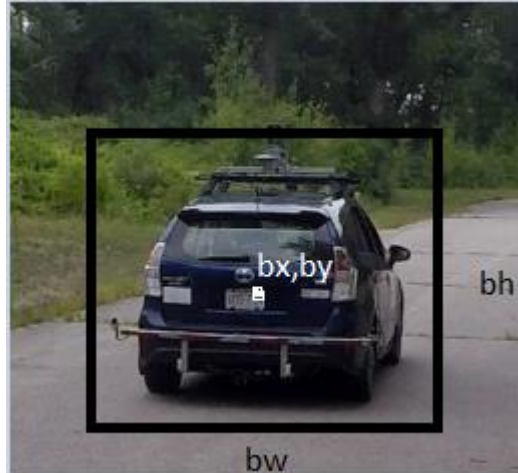


13.1 Localization:

Localization is the task of finding where the object is located in the input image and drawing a bounding box around it. So localization first does the classification job of finding whether there is an instance of any class or not and then draws the bounding box around it. Usually there is one (or any other known number) big object in the center (as opposed to object detection).

Taking the example of a self-driving car the localization pipeline is as follows:
Let's suppose we decide the following object categories for the self-driving car :
Pedestrian, Car and Bike. Also if there is no object from these three classes then classify image as background

So this is the standard classification pipeline. How about if you want to localize the car in the image as well. To do that, you can change your neural network to have a few more output units that output a bounding box. So, in particular, you can have the neural network output four more numbers, and I'm going to call them b_x , b_y , b_h , and b_w . And these four numbers parameterized the bounding box of the detected object.
The standard neural network is changed to have a few more output units that output a bounding box. The bounding box is specified by four numbers: b_x , b_y , b_h , b_w respectively the box center (x,y) coordinates and box height and width.



13.2 Detection:

Object Detection involves drawing a box around all instances of the class in the image. There can be more than one class and more than one object per class. Number of instances of a class is not known in advance. This is for example used in self driving cars for real time detection of pedestrians, cars, trucks, signal, etc. on the road.

The input image is divided into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts. Formally we define confidence as:

$$\text{Pr}(\text{Object}) * \text{IOU}_{\text{truth_pred}}$$

If no object exists in that cell, the confidence scores should be zero. Otherwise the confidence score should equal the intersection over union (IOU) between the predicted box and the ground truth.






Each bounding box consists of 5 predictions: x , y , w , h , and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally the confidence prediction represents the IOU between the predicted box and any ground truth box.

13.2.1 Sliding Windows Detection Algorithm

Suppose we are creating a car detection algorithm. The pipeline for this task can be:

1. Build a labelled training set. For our purposes we can start with closely cropped centered images. (meaning x is pretty much only car). Lets say the input image size is (14x14).

2. Given this set, train the ConvNet to classify whether an image is car or not.

| x | y |
|---|---|
|  | 1 |
|  | 1 |
|  | 1 |
|  | 0 |
|  | 0 |

3. Detection often requires fine-grained visual information so at test time the input image resolution is high, let's say it is (28x28). So we pass a cropped square from input image through the ConvNet at test time and let it output 0 or 1. Shift the square sideways until every region of the image is passed through the ConvNet.

4. Choose a larger window size and repeat.

5. Hope is that if there is a car somewhere in the image there will be a window passing which through the ConvNet will output 1 and detect the car is located there.



2 different window sizes

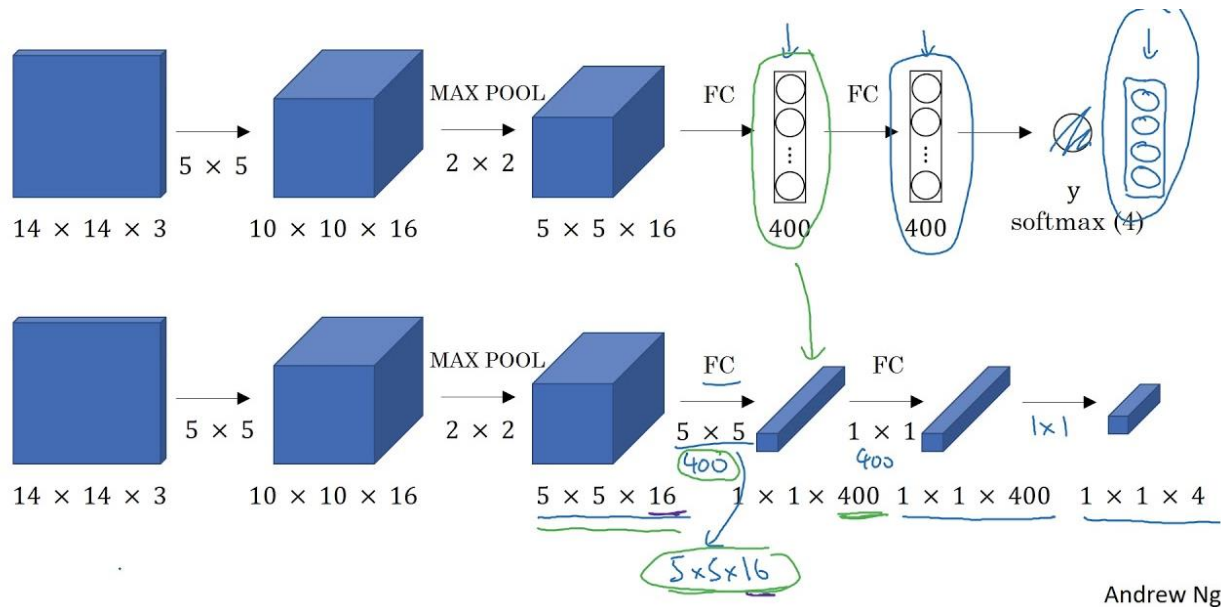
Disadvantage of SWA:

- Computationally expensive
- Time consuming

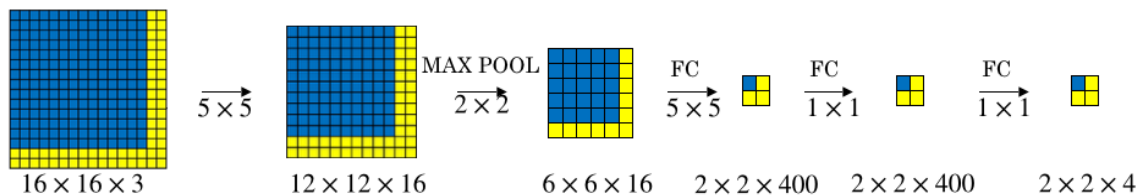
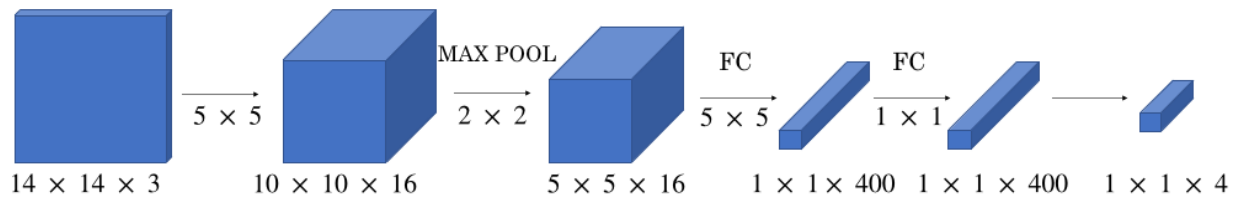
13.2.2 Convolutional Implementation of Sliding Windows

Algorithm:

We convert the fully connected layers into convolutional layers as shown



Andrew Ng



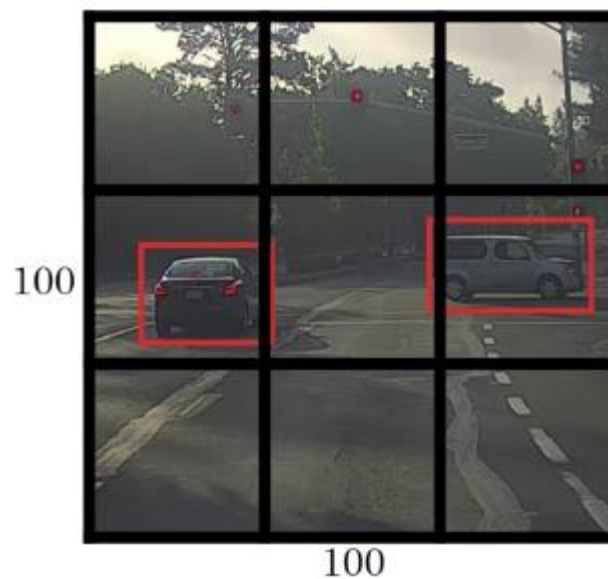
We apply convolutional techniques to the whole image and output a $2 \times 2 \times 4$ image. Each $1 \times 1 \times 4$ in the output corresponds to one of the sliding windows as shown in different colors above. Moreover, this convolutional replacement of the sliding window protocol is much more economical.

Now, this algorithm still has one weakness, the position of the bounding boxes is not too accurate.

13.2.3 YOLO Algorithm – You Only Look Once

“You Only Look Once” (YOLO) is a popular algorithm because it achieves high accuracy while also being able to run in real-time. It is extremely fast. This algorithm “only looks once” at the image in the sense that it requires only one forward propagation pass through the network to make predictions. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

- To understand the above algorithm, we take a 100×100 image and place 3×3 grid cells over it.



- We take image classification and localization algorithm & apply that to each of the nine grid cells of this image.

- Now, our labels would include $p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3$. The output will be $[p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]^T$. p_c represents whether or not there is an instance of any class or is it just background. c_i represents the probability that an object from class i is present in the grid. And $(^T)$ is the transpose so that it is a column vector
- Label for upper left grid cell would be

$$\begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$

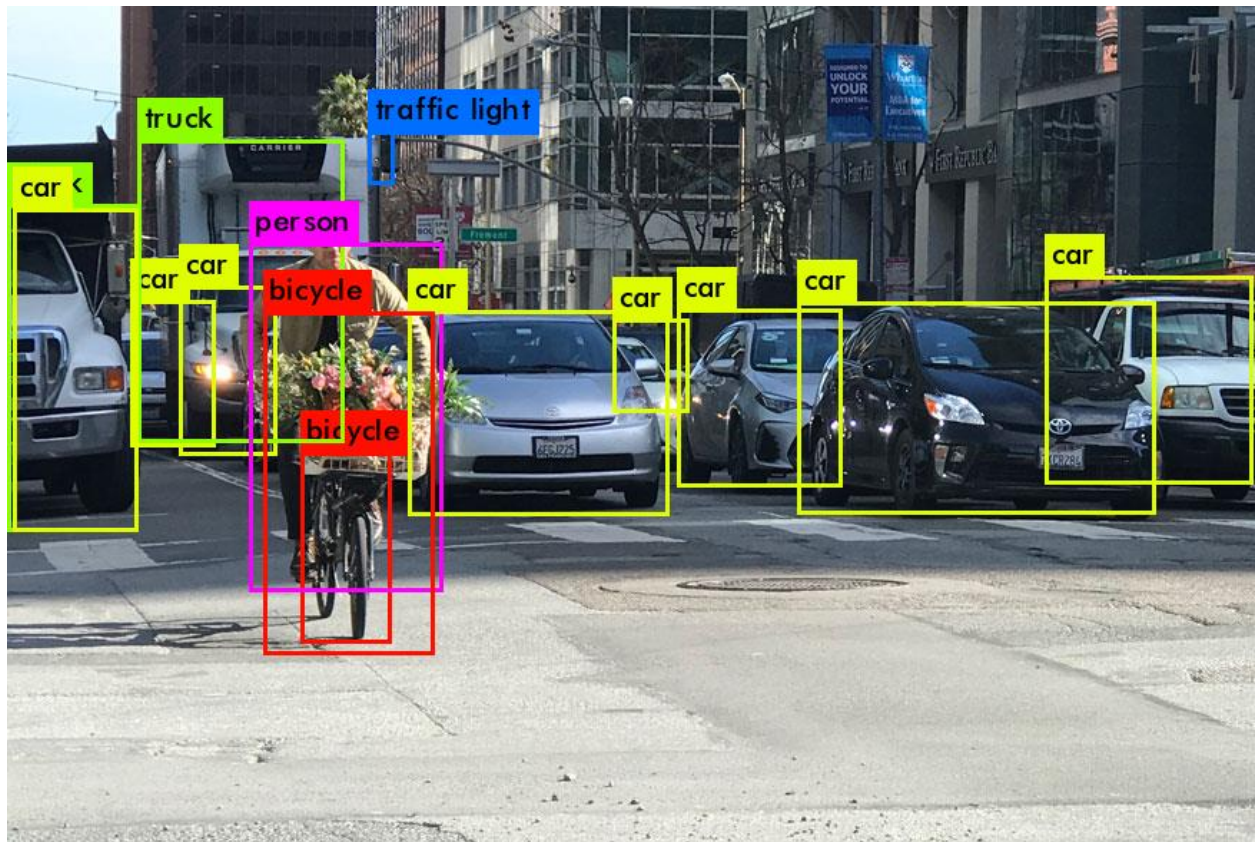
? represents don't care since there is just background

- As per the Yolo algorithm, the object is assigned to the grid cell whose midpoint lies in it. Therefore, the first car object is assigned to the 4th grid cell and the second car object is assigned to the 6th grid cell.
- The target output in the above case would be $3 \times 3 \times 8$ in correspondence to each grid cell.
- The bounding box for the second car would be

$$\begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{matrix} 0.4 \\ 0.3 \\ 0.9 \\ 0.5 \end{matrix}$$

b_x and b_y would be smaller than 1 as they represent the grid cell where the midpoint of the object lies whereas b_h and b_w could be larger than 1 as the image could be present in more than 1 grid cell.

This is how the predicted bounding boxes look after YOLO is applied



Link to the Assignments

Assignment-3 of CS231n course and Assignments of Coursera Course

<https://github.com/purvi0116/Neural-Networks-Assignments>

REFERENCES

[Neural networks and deep learning](#)

[Convolutional Neural Networks for Visual Recognition](#)

[Deeplearning.ai](#)

[Deep Learning with PyTorch: A 60 Minute Blitz — PyTorch Tutorials 1.5.0 documentation](#)

<https://towardsdatascience.com/>

[ImageNet Classification with Deep Convolutional Neural Networks](#)

[Visualizing and Understanding Convolutional Networks](#)

[Deep Residual Learning for Image Recognition](#)

[A Neural Algorithm of Artistic Style](#)

What all have I learnt and how I went about it?

I started with the Neural Networks book by Michael Neilson which helped me get familiar with what Neural Network is .

Then I completed the CS231n lecture series and assignments by Stanford University. I wanted to know more about Object Localization and Detection and found the Coursera Course- Convolutional Neural Networks by Andrew NG (Course 4 of the specialization) which explains the application Neural Networks in various state-of-the-art computer vision tasks. I also read a few research papers namely Alexnet, ZFNet, ResNet, Neural Style Transfer and YOLO.

It was a great learning experience this summer where I learnt how the present day technologies use Machine Learning and specifically Neural Networks (or Deep Learning) to provide us the best services.