ECE196 Face Recognition

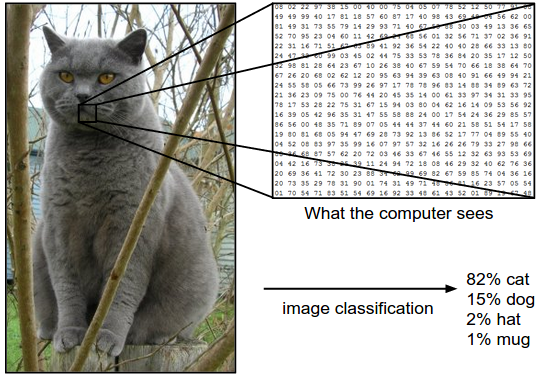
Notes on Convolutional Neural Networks

Adapted from <http://cs231n.github.io/>

Will Chen

# Image Classification Problems

An image Classification problem is the task of assigning an input image one label from a fixed set of categories. For example, in the image below an image classification model takes a single image and assigns probabilities to 4 labels, {cat, dog, hat, mug}. As shown in the image, keep in mind that to a computer an image is represented as one large 3-dimensional array of numbers. In this example, the cat image is 248 pixels wide, 400 pixels tall, and has three color channels Red, Green, Blue (or RGB for short). Therefore, the image consists of 248 x 400 x 3 numbers, or a total of 297,600 numbers. Each number is an integer that ranges from 0 (black) to 255 (white). Our task is to turn these numbers into a single label, such as “cat”.



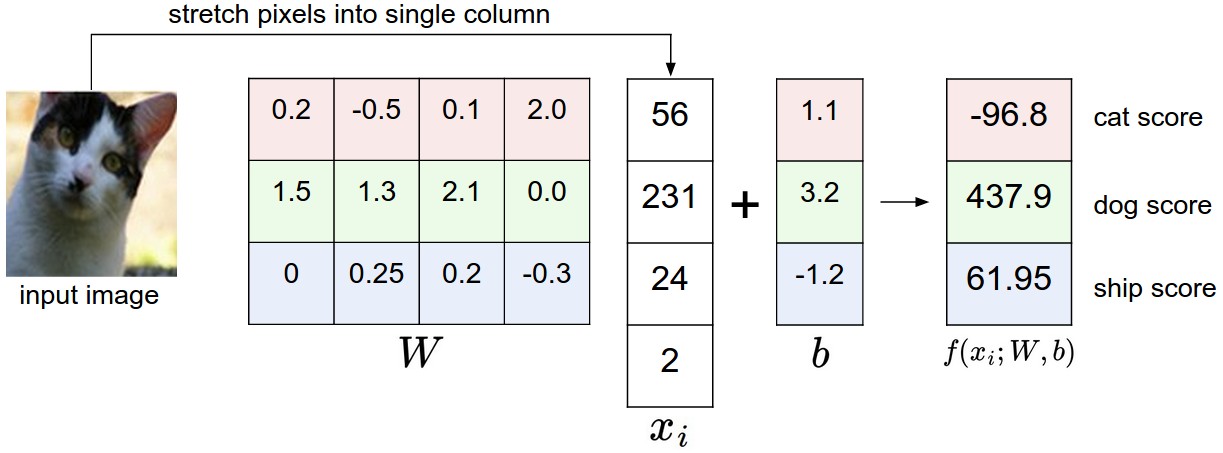
# Linear Classification

We are now going to develop a basic classifier that will eventually extend to entire Neural Networks and Convolutional Neural Networks. The approach will have two major components: a score function that maps the raw data to class scores, and a loss function that quantifies the deviation between the predicted scores and the ground truth labels. We will then cast this as an optimization problem in which we will minimize the loss function with respect to the parameters of the score function.

Let’s start with a linear function f(xi,W,b)=Wxi+b, where xi has all of its pixels flattened out to a single column vector of shape [D x 1], and D is the number of pixels in an image. The matrix W (of size [K x D]), and the vector b (of size [K x 1]) are the parameters of the function, where K is the number of classes. The W and b are usually referred to as weights or parameters.

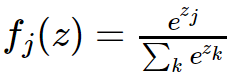
We wish that the correct class has a score that is higher than the scores of incorrect classes. Convolutional Neural Networks will map image pixels to scores exactly as shown above, but the mapping (f) will be more complex and will contain more parameters.

Below is an example of using an linear classifier to map an image to class scores. For the sake of visualization, we assume the image only has 4 pixels, and that we have 3 classes (red (cat), green (dog), blue (ship) class). (Clarification: in particular, the colors here simply indicate 3 classes and are not related to the RGB channels.) We stretch the image pixels into a column and perform matrix multiplication to get the scores for each class. Note that this particular set of weights W is not good at all: the weights assign our cat image a very low cat score. In particular, this set of weights seems convinced that it's looking at a dog.

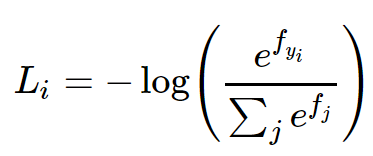


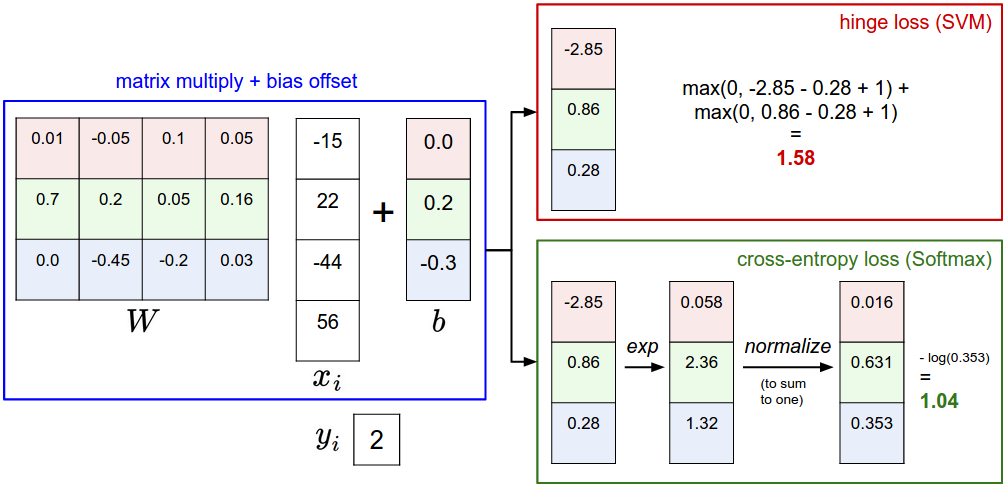
# Loss Function

A loss function represents how much the predicted results deviate from the ground truth. If predicted results are the same as ground truth, the loss function should give 0, whereas if they are very different, the loss function should give a large number.

Here I’d live to introduce a classifier called Softmax. It is one of the most commonly used classifier in neural networks. Softmax is expressed as below, where we are using the notation fj to mean the j-th element of the vector of class scores f.  
 , 

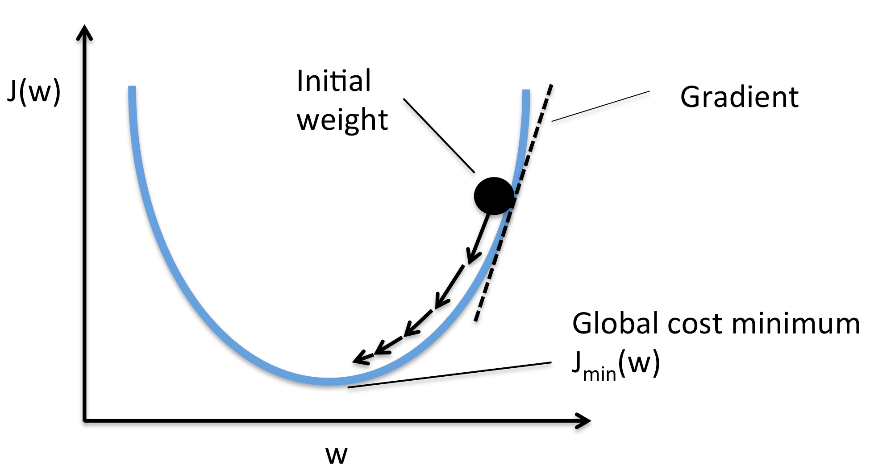
The loss function corresponding to Softmax is:





# Gradient Descent

We are now going to introduce the third and last key component: optimization. Optimization is the process of finding the set of parameters W that minimize the loss function, which indicates accurate predictions. When we look at loss functions with respect to W, the loss functions are usually convex functions. Hence, we can find the best W using an iterative procedure by following the gradient. In a hiking analogy, this approach roughly corresponds to feeling the slope of the hill below our feet and stepping down the direction that feels steepest. An illustration is shown below in the plot.



<https://sebastianraschka.com/images/faq/closed-form-vs-gd/ball.png>

Gradient can be computed as . The parameters can be updated using the formula below, where is the learning rate. .

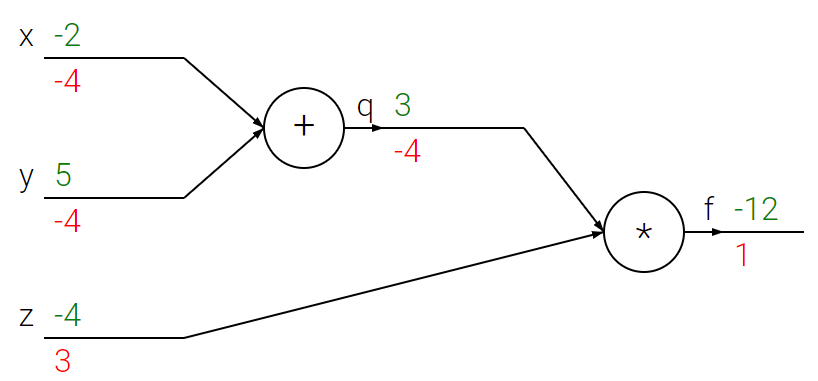
A common technique used in neural networks is Stochastic Gradient Descent (SGD). It usually refers to the process of evaluating on a small batch of samples, based on the assumption that the small batch is a good approximation of the entire dataset. This method allows faster convergence and saves computational cost.

# Backpropagation

From the last section, we learned how parameters are updated. The process can be very complex when we deal with deep neural networks. In this section, we will develop an intuitive understanding of backpropagation, which is a way of computing gradients of expressions through recursive application of chain rule. In this way, gradient can be computed step by step.

Let’s now start to consider more complicated expressions that involve multiple composed functions, such as f(x,y,z)=(x+y)zf(x,y,z)=(x+y)z. This expression is still simple enough to differentiate directly, but we’ll take a particular approach to it that will be helpful with understanding the intuition behind backpropagation. In particular, note that this expression can be broken down into two expressions: q=x+yq=x+y and f=qzf=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 , and q is addition of x and y so . However, we don’t necessarily care about the gradient on the intermediate value q - the value of 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, . In practice, this is simply a multiplication of the two numbers that hold the two gradients.

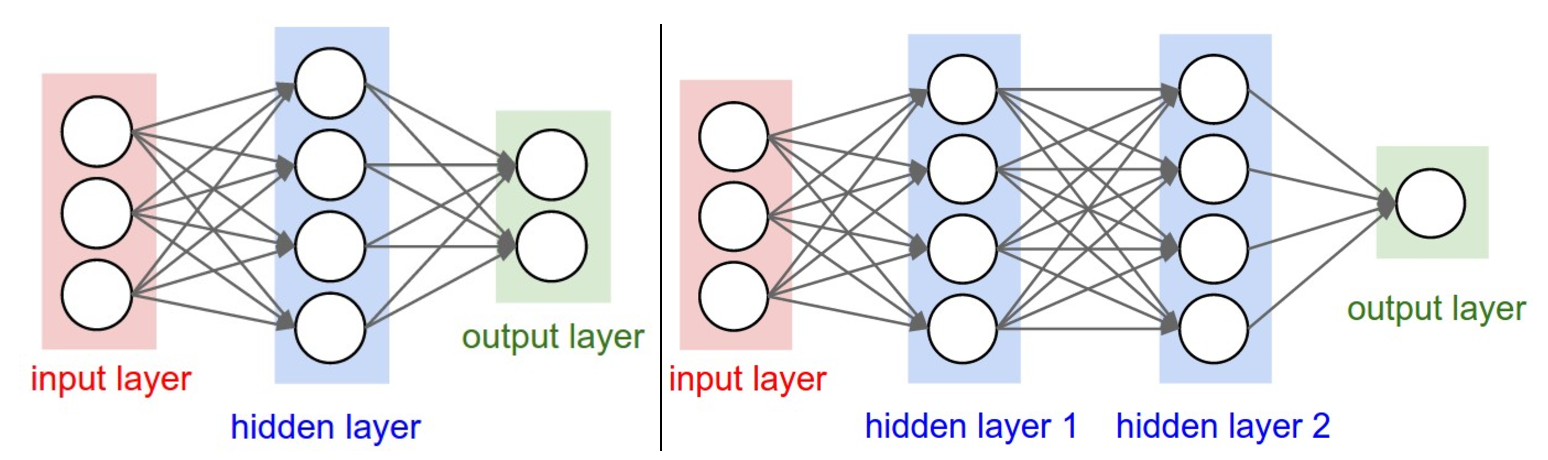
The diagram below shows the visual representation of the computation. The forward pass computes values from inputs to output (shown in green). The backward pass then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the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



# Convolutional Neural Network Architecture

## Fully Connected Layers

Neurons (represented as circles in the diagram below) between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. Each neuron represents one parameter, represented as W in the previous discussions. Below are two example Neural Network topologies that use a stack of fully-connected layers:



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections between neurons across layers, but not within a layer.

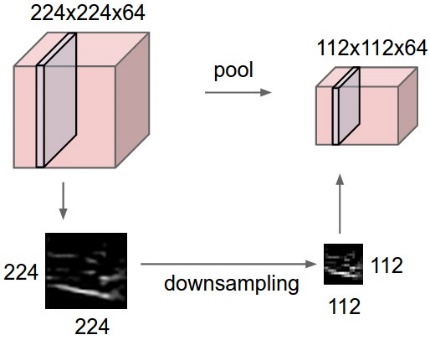
## Convolutional Layers

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. A 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 5x5x3 (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 of how CONV layers work can be found on this page: <http://cs231n.github.io/convolutional-networks/>. Search “Convolution Demo” on the page.

we must specify the stride with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.

## Pooling Layers

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 number of parameters and computation in the network, and hence to also control overfitting. The most common form is a pooling layer with filters of size 2x2, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged. An example is show below.

## Activation Functions

Activation functions introduce non-linearity to neural networks. There are several activation functions you may encounter in practice.

|  |  |  |
| --- | --- | --- |
| Sigmoid: squashes real numbers to range between [0,1] | Tanh: squashes a real-valued number to the range [-1, 1] | ReLU: the activation is simply thresholded at zero |
|  |  |  |

# Transfer Learning

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

One common technique is finetuning. Finetuning is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet, or it’s possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a ConvNet contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset. In case of ImageNet for example, which contains many dog breeds, a significant portion of the representational power of the ConvNet may be devoted to features that are specific to differentiating between dog breeds.