Summer of Science Report

Computer Vision

Submitted by

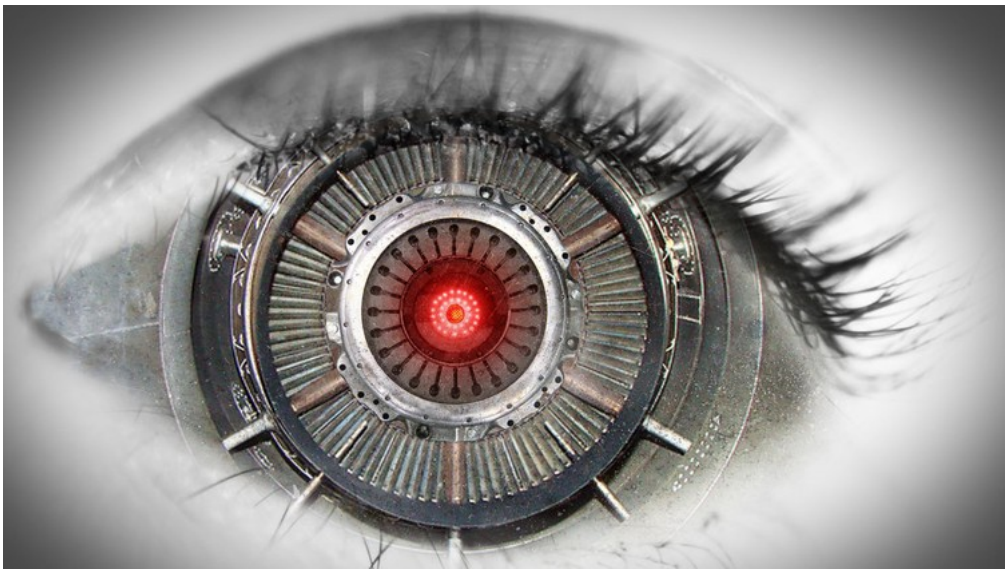
Karan Agarwalla
180050045
Undergrad at
Indian Institute of Technology Bombay

Under the guidance of

Sarthak Consul

IIT BOMBAY

July 2019



1 Introduction

Machine learning is the science of getting computers to act without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech recognition, effective web search, and a vastly improved understanding of the human genome. Machine learning is so pervasive today that you probably use it dozens of times a day without knowing it.

Computer vision is an interdisciplinary field that deals with how computers can be made to gain high-level understanding from digital images or videos. From the perspective of engineering, it seeks to automate tasks that the human visual system can do. It involves the development of a theoretical and algorithmic basis to achieve automatic visual understanding.As a scientific discipline, computer vision is concerned with the theory behind artificial systems that extract information from images. In this project, we learnt about **Convolutional Neural Networks** and their application in Computer Vision.

Acknowledgment

This work was written as a part of the Summer Of Science, 2019 by the Maths and Physics Club, IIT Bombay. I have followed the course CS 231N Convolutional Neural Networks for Visual Recognition by Stanford University. The course instructors were Prof. Fei-Fei Li, Justin Johnson and Serena Yeung. All the content here is highly inspired by this course and at many points might simply be a paraphrasing of the course content. Nevertheless, I have tried to compress the contents of the course, skipping a few of the details, while still preserving sufficient depth. Also, I would like to acknowledge the help of my guide, Sarthak Consul, for his help and support. I hope that this work helps you in some way, just as it has helped me learn the basics Deep Learning and Image Processing.

Contents

2 Image Classification

2.1 Motivation

We will introduce the Image Classification problem, which is the task of assigning an input image one label from a fixed set of categories. This is one of the core problems in Computer Vision that, despite its simplicity, has a large variety of practical applications.

2.2 Challenges

Since this task of recognizing a visual concept is relatively trivial for a human to perform, it is worth considering the challenges involved from the perspective of a Computer Vision algorithm. Keeping in mind the raw representation of images as a 3-D array of brightness values,the challenges are as follows:

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.

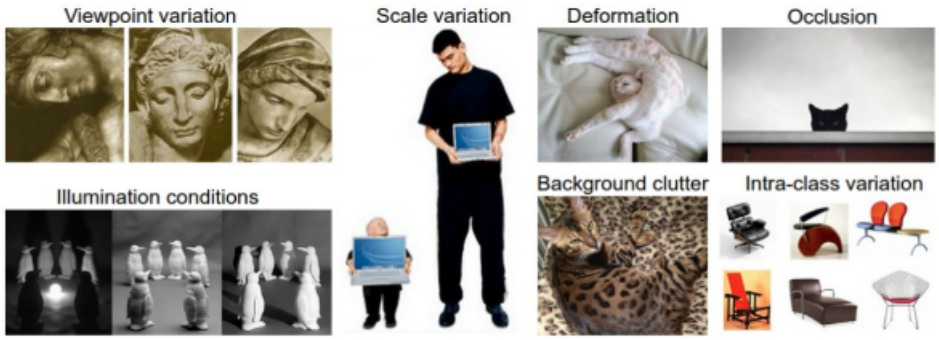
Background clutter

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.

A good image classification model must be invariant to the cross product of all these variations, while simultaneously retaining sensitivity to the inter-class variations.



2.3 Problem Specifications

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 that it has never seen before. We will

then compare the true labels of these images to the ones predicted by the classifier. Intuitively, were hoping that a lot of the predictions match up with the true answers (which we call the ground truth).

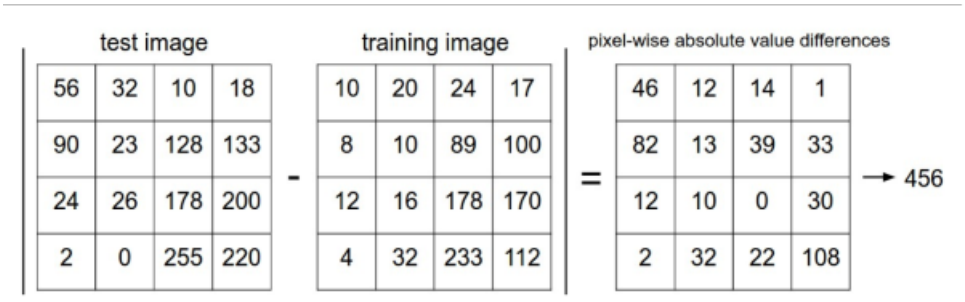
3 Nearest Neighbour Classifier

3.1 Nearest Neighbour Classifier

We have to compare two images. One of the simplest possibilities is to compare the images pixel by pixel and add up all the differences. In other words, given two images and representing them as vectors I1,I2 , a reasonable choice for comparing them might be the L1 distance:

$$d_1(I_1, I_2) = \sum_{p=1}^{n} |I_1^p - I_2^p|$$

Where the sum is taken over all pixels. Here is the procedure visualized:



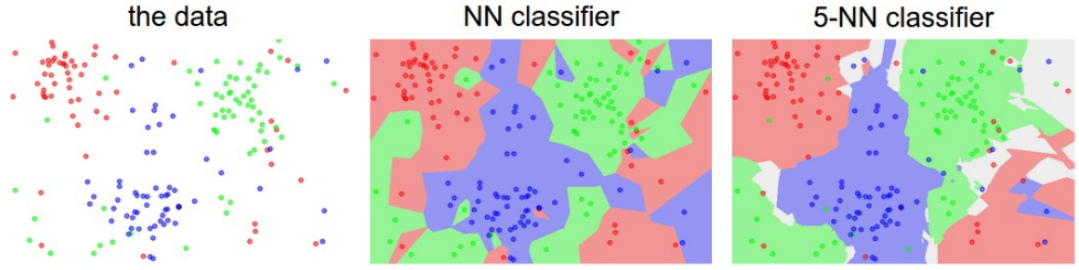
An example of using pixel-wise differences to compare two images with L1 distance (for one color channel in this example). Two images are subtracted elementwise and then all differences are added up to a single number. If two images are identical the result will be zero. But if the images are very different the result will be large.

Choice of distance

here are many other ways of computing distances between vectors. Another common choice could be to instead use the L2 distance, which has the geometric interpretation of computing the euclidean distance between two vectors. The distance takes the form: $d_2(I_1, I_2) = \sum_{p=1}^{n} \sqrt{(I_1^p - I_2^p)^2}$

3.2 k-Nearest Neighbour Classifier

It is strange to only use the label of the nearest image when we wish to make a prediction. Indeed, it is almost always the case that one can do better by using whats called a k-Nearest Neighbor Classifier. The idea is very simple: 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:



An example of the difference between Nearest Neighbor and a 5-Nearest Neighbor classifier, using 2-dimensional points and 3 classes (red, blue, green). The colored regions show the **decision boundaries** induced by the classifier with an L2 distance. The white regions show points that are ambiguously classified (i.e. class votes are tied for at least two classes). Notice that 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, likely leading to better **generalization** on the test data (not shown). Also note that the gray regions in the 5-NN image are caused by ties in the votes among the nearest neighbors (e.g. 2 neighbors are red, next two neighbors are blue, last neighbor is green).

3.3 Validation Sets

The k-nearest neighbor classifier requires a setting for k. But what number works best? Additionally, we see that there are many different distance functions we could have used: L1 norm, L2 norm, there are many other choices we didn't even consider. These choices are called hyperparameters and they come up very often in the design of many Machine Learning algorithms that learn from data. Its often not obvious what values/settings one should choose.

You might be tempted to suggest that we should try out many different values and see what works best. That is a fine idea and thats indeed what we will do, but this must be done very carefully. In particular, we cannot use the test set for the purpose of tweaking hyperparameters. Whenever youre designing Machine Learning algorithms, you should think of the test set as a very precious resource that should ideally never be touched until one time at the very end. Otherwise, the very real danger is that you may tune your hyper parameters to work well on

the test set, but if you were to deploy your model you could see a significantly reduced performance. In practice, we would say that you overfit to the test set.

Luckily, there is a correct way of tuning the hyperparameters and it does not touch the test set at all. The idea is to split our training set in two: a slightly smaller training set, and what we call a validation set.

Cross-validation

In cases where the size of your training data (and therefore also the validation data) might be small, people sometimes use a more sophisticated technique for hyperparameter tuning called cross-validation. Working with our previous example, the idea is that instead of arbitrarily picking the first 1000 datapoints to be the validation set and rest training set, you can get a better and less noisy estimate of how well a certain value of k works by iterating over different validation sets and averaging the performance across these. For example, in 5-fold cross-validation, we would split the training data into 5 equal folds, use 4 of them for training, and 1 for validation. We would then iterate over which fold is the validation fold, evaluate the performance, and finally average the performance across the different folds.

3.4 Disadvantages

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.

4 Linear Classification

The linear classification approach will have two major components: a **score function** that maps the raw data to class scores, and a **loss function** that quantifies the agreement 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.

4.1 Score function

$$f(x_i, W, b) = Wx_i + b$$

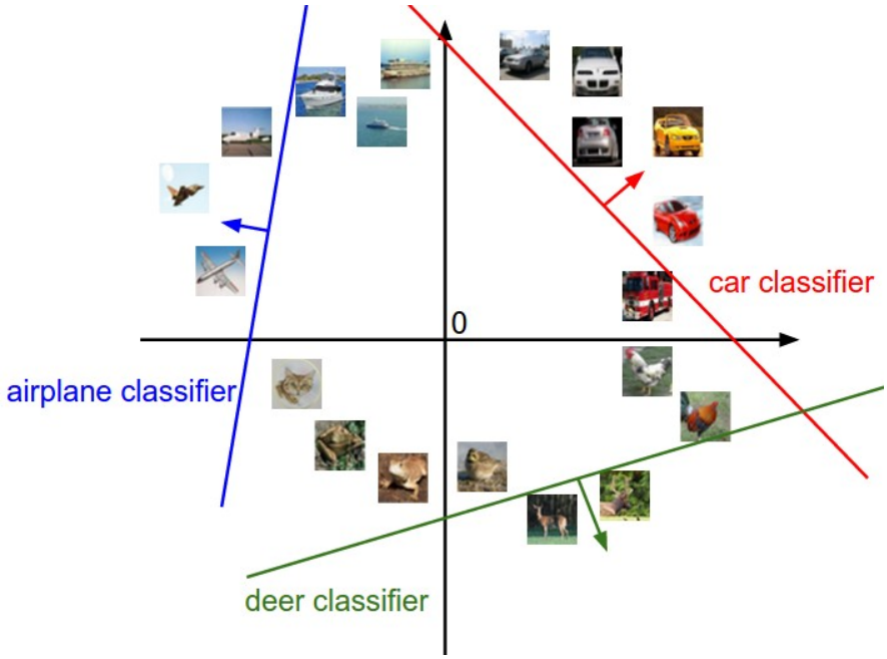
In the above equation, we are assuming that the image x_i has all of its pixels flattened out to a single column vector of shape [D x 1]. The matrix W (of size [K x D]), and the vector b (of size [K x 1]) are the parameters of the function. In CIFAR-10, x_i contains all pixels in the i-th image flattened into a single [3072 x 1] column, W is [10 x 3072] and b is [10 x 1], so 3072 numbers come into the function (the raw pixel values) and 10 numbers come out (the class scores). The parameters in W are often called the **weights**, and b is called the **bias vector** because it influences the output scores, but without interacting with the actual data x_i.
First, note that the single matrix multiplication Wx_i is effectively evaluating 10 separate classifiers in parallel (one for each class), where each classifier is a row of W.
We think of the input data (x_i,y_i) as given and fixed, but we have control over the setting of the parameters W,b. Our goal will be to set these in such way that the computed scores match the ground truth labels across the whole training set.We wish that the correct class has a score that is higher than the scores of incorrect classes.
An advantage of this approach is that the training data is used to learn the parameters W,b, but once the learning is complete we can discard the entire training

set and only keep the learned parameters. That is because a new test image can be simply forwarded through the function and classified based on the computed scores.
Lastly, note that classifying the test image involves a single matrix multiplication and addition, which is significantly faster than comparing a test image to all training images.
Depending on precisely what values we set for these weights, the function has the capacity to like or dislike (depending on the sign of each weight) certain colors at certain positions in the image.

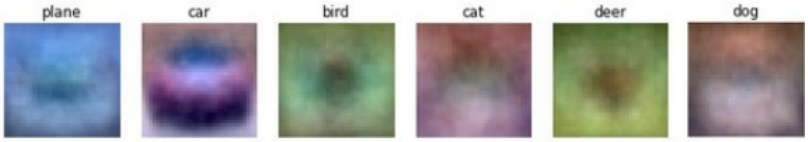
4.2 Interpretation of Linear Classifiers

Analogy of images as high-dimensional points Since the images are stretched into high-dimensional column vectors, we can interpret each image as a single point in this space (e.g. each image in CIFAR-10 is a point in 3072-dimensional space of 32x32x3 pixels). Analogously, the entire dataset is a (labeled) set of points.

Since we defined the score of each class as a weighted sum of all image pixels, each class score is a linear function over this space. We cannot visualize 3072-dimensional spaces, but if we imagine squashing all those dimensions into only two dimensions, then we can try to visualize what the classifier might be doing.



Interpretation of linear classifiers as template matching. Another interpretation for the weights **W** is that each row of **W** corresponds to a template (or sometimes also called a prototype) for one of the classes. The score of each class for an image is then obtained by comparing each template with the image using an inner product (or dot product) one by one to find the one that fits best. With this terminology, the linear classifier is doing template matching, where the templates are learned. Another way to think of it is that we are still effectively doing Nearest Neighbor, but instead of having thousands of training images we are only using a single image per class (although we will learn it, and it does not necessarily have to be one of the images in the training set), and we use the (negative) inner product as the distance instead of the L1 or L2 distance.
However,The linear classifier is too weak to properly account for different-colored cars.



Bias Trick

As we proceed through the material it is a little cumbersome to keep track of two sets of parameters (the biases b and weights W) separately. A commonly used trick is to combine the two sets of parameters into a single matrix that holds both of them by extending the vector xi with one additional dimension that always holds the constant 1 - a default bias dimension. With the extra dimension, the new score function will simplify to a single matrix multiply:

$$f(x_i, W, b) = Wx_i + b$$

Image Data Preprocessing

In particular, it is important to center your data by subtracting the mean from every feature. In the case of images, this corresponds to computing a mean image across the training images and subtracting it from every image to get images where the pixels range from approximately [-127 127]. Further common preprocessing is to scale each input feature so that its values range from [-1, 1].

4.3 Loss Function

In the previous section we defined a function from the pixel values to class scores, which was parameterized by a set of weights **W**. Moreover, we saw that we dont have control over the data (x_i,y_i) (it is fixed and given), but we do have control over these weights and we want to set them so that the predicted class scores are consistent with the ground truth labels in the training data.Intuitively, the loss will be high if were doing a poor job of classifying the training data, and it will be low if were doing well.

4.3.1 Multiclass Support Vector Machine loss

The SVM loss is set up so that the SVM wants the correct class for each image to a have a score higher than the incorrect classes by some fixed margin . The score function takes the pixels and computes the vector f(xi,W) of class scores, which we will abbreviate to s (short for scores). 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:

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

The threshold at zero **max(0,)** function is often called the **hinge loss**. The squared hinge loss SVM (or L2-SVM) $\boldsymbol{(max(0,-)^2}$ is sometimes used. The loss function quantifies our unhappiness with predictions on the training set.

Regularization

Suppose that we have a dataset and a set of parameters W that correctly classify every example (i.e. all scores are so that all the margins are met, and L_i=0 for all i). The issue is that this set of W is not necessarily unique: there might be many similar W that correctly classify the examples. One easy way to see this is that if some parameters W correctly classify all examples (so loss is zero for each example), then any multiple of these parameters W where ¿1 will also give zero loss because this transformation uniformly stretches all score magnitudes and hence also their absolute differences. We wish to encode some preference for a certain set of weights W over others to remove this ambiguity. We can do so by extending the loss function with a **regularization penalty R(W)**. The most common regularization penalty is the **L2 norm** that discourages large weights through an element-wise quadratic penalty over all parameters:

$$R(W) = \sum_{k=1} \sum_{l=1} W_{k,l}^2$$

In the expression above, we are summing up all the squared elements of W. Notice that the regularization function is not a function of the data, it is only based on the weights. Including the regularization penalty completes the full Multiclass Support Vector Machine loss, which is made up of two components: the **data loss** (which is the average loss Li over all examples) and the **regularization loss**. That is, the full Multiclass SVM loss becomes:

$$L = \frac{\sum_{i=1} L_i}{N} + \lambda R(W)$$

we append the regularization penalty to the loss objective, weighted by a hyper-parameter .There is no simple way of setting this hyperparameter and it is usually determined by cross-validation.

4.3.2 Softmax Classifier

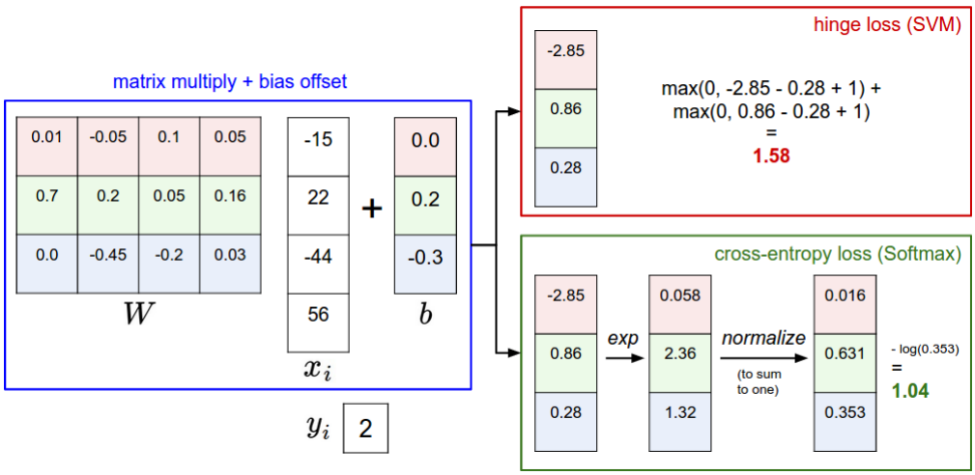
Unlike the SVM which treats the outputs $f(x_i, W)$ as (uncalibrated and possibly difficult to interpret) scores for each class, the Softmax classifier gives a slightly more intuitive output (normalized class probabilities) and also has a probabilistic interpretation that we will describe shortly. In 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 and replace the hinge loss with a cross-entropy loss that has the form:

$$\boldsymbol{L_i = -\log(} \frac{\boldsymbol{e^{f_{y_i}}}}{\boldsymbol{\sum_j e^{f_{y_j}}}}\boldsymbol{)}$$

where we are using the notation $\boldsymbol{f_j}$ to mean the j-th element of the vector of class scores f.

4.3.3 SVM vs Softmax

Unlike the SVM which computes uncalibrated and not easy to interpret scores for all classes, the Softmax classifier allows us to compute probabilities for all labels.So, SVM may not differentiate in case of wide differences giving exact 0 whereas Softmax prefers wider differences. Compared to the Softmax classifier, the SVM is a more local objective, which could be thought of either as a bug or a feature. In other words, the Softmax classifier is never fully happy with the scores it produces: the correct class could always have a higher probability and the incorrect classes always a lower probability and the loss would always get better. However, the SVM is happy once the margins are satisfied and it does not micromanage the exact scores beyond this constraint. However, the performance difference between the SVM and Softmax are usually very small.

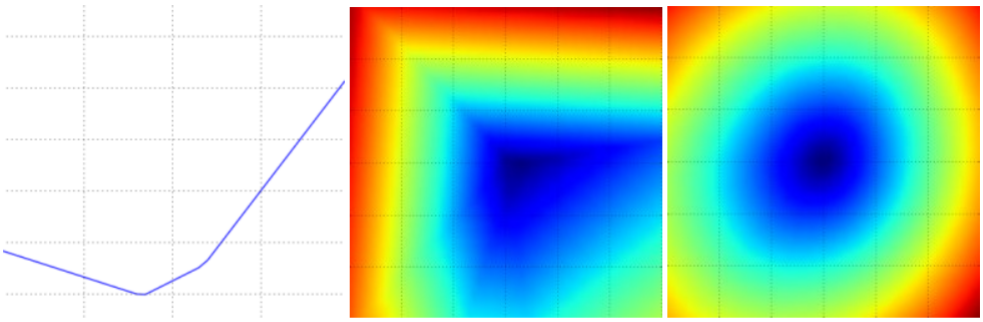

matrix multiply + bias offset
0.01
-0.05
0.1
0.05
0.7
0.2
0.05
0.16
0.0
-0.45
-0.2
0.03
W
-15
22
-44
56
x_i
+
0.0
0.2
-0.3
b
y_i
2
hinge loss (SVM)
-2.85
0.86
0.28
max(0, -2.85 - 0.28 + 1) +
max(0, 0.86 - 0.28 + 1)
=
1.58
cross-entropy loss (Softmax)
-2.85
0.86
0.28
exp
0.058
2.36
1.32
normalize
(to sum
to one)
0.016
0.631
0.353
- log(0.353)
=
1.04

5 Optimisation

Optimization is the process of finding the set of parameters W that minimize the loss function.

5.1 Visualising the loss function

The loss functions well look at in this class are usually defined over very high-dimensional spaces (e.g. in CIFAR-10 a linear classifier weight matrix is of size [10 x 3073] for a total of 30,730 parameters), making them difficult to visualize. However, we can still gain some intuitions about one by slicing through the high-dimensional space along rays (1 dimension), or along planes (2 dimensions).



Loss function landscape for the Multiclass SVM (without regularization) for one single example (left,middle) and for a hundred examples (right) in CIFAR-10. Left: one-dimensional loss by only varying **a**. Middle, Right: two-dimensional loss slice, Blue = low loss, Red = high loss. Notice the piecewise-linear structure of the loss function. The losses for multiple examples are combined with average, so the bowl shape on the right is the average of many piece-wise linear bowls (such as the one in the middle).

5.2 Strategies

Random search

Since it is so simple to check how good a given set of parameters W is, the first (very bad) idea that may come to mind is to simply try out many different random

weights and keep track of what works best.

Random Local Search

The first strategy you may think of is to try to extend one foot in a random direction and then take a step only if it leads downhill. Concretely, we will start out with a random W, generate random perturbations W to it and if the loss at the perturbed W+W is lower, we will perform an update.

Following the Gradient

We can compute the best direction along which we should change our weight vector that is mathematically guaranteed to be the direction of the steepest descend (at least in the limit as the step size goes towards zero). This direction will be related to the gradient of the loss function.

Computing the Gradient

Numerical Gradient
The code above iterates over all dimensions one by one, makes a small change h along that dimension and calculates the partial derivative of the loss function along that dimension by seeing how much the function changed.We update in the negative direction.
Effect of step size The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step. As we will see later in the course, choosing the step size (also called the learning rate) will become one of the most important (and most headache-inducing) hyperparameter settings in training a neural network.

Analytical Gradient The numerical gradient is very simple to compute using the finite difference approximation, but the downside is that it is approximate (since we have to pick a small value of h, while the true gradient is defined as the limit as h goes to zero), and that it is very computationally expensive to compute. The second way to compute the gradient is analytically using Calculus, which allows us to derive a direct formula for the gradient (no approximations) that is also very fast to compute. However, unlike the numerical gradient it can be more error prone to implement, which is why in practice it is very common to compute the analytic gradient and compare it to the numerical gradient to check the correctness of your implementation. This is called a **gradient check**.

5.3 Gradient Descent

Now that we can compute the gradient of the loss function, the procedure of repeatedly evaluating the gradient and then performing a parameter update is called Gradient Descent. Its vanilla version looks as follows:

$$weights+ = -stepsize * weightsgrad$$

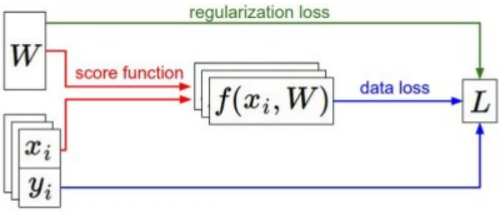
Mini Batch Gradient Descent

In large-scale applications (such as the ILSVRC challenge), the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data.

Stochastic Gradient Descent (SGD)

The extreme case of this is a setting where the mini-batch contains only a single example. This process is called Stochastic Gradient Descent (SGD) (or also sometimes on-line gradient descent). This is relatively less common to see because in practice due to vectorized code optimizations it can be computationally much more efficient to evaluate the gradient for 100 examples, than the gradient for one example 100 times.The size of the mini-batch is a hyperparameter but it is not very common to cross-validate it. It is usually based on memory constraints (if any), or set to some value, e.g. 32, 64 or 128. We use powers of 2 in practice because many vectorized operation implementations work faster when their inputs are sized in powers of 2.

5.4 Summary



Summary of the information flow. The dataset of pairs of **(x,y)** is given and fixed. The weights start out as random numbers and can change. During the forward pass the score function computes class scores, stored in vector **f**. The loss function contains two components: The data loss computes the compatibility between the scores **f** and the labels **y**. The regularization loss is only a function of the weights. During Gradient Descent, we compute the gradient on the weights (and optionally on data if we wish) and use them to perform a parameter update during Gradient Descent.

6 Back Propagation

Backpropagation is a way of computing gradients of expressions through recursive application of **chain rule**. Let f(x,y,z)=(x+y)z. This expression is still simple enough to differentiate directly, but well 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+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 $\partial f/\partial q=z$,$\partial f/\partial z=q$, and q is addition of x and y so $\partial q/\partial x=1$,$\partial q/\partial y=1$. However, we dont necessarily care about the gradient on the intermediate value q - the value of $\partial f/\partial q$ 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, $\partial f/\partial x=(\partial f/\partial q)(\partial q/\partial x)$. In practice this is simply a multiplication of the two numbers that hold the two gradients.

6.1 Intuitive Understanding of Backpropagation

Notice that 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 inputs with respect to its output value. 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.
The **and gate** always takes the gradient on its output and distributes it equally to all of its inputs, regardless of what their values were during the forward pass.
The **max gate** routes the gradient. Unlike the add gate which distributed the gradient unchanged to all its inputs, the max gate distributes the gradient (un-

changed) to exactly one of its inputs (the input that had the highest value during the forward pass).
The **multiply gate** is a little less easy to interpret. Its local gradients are the input values (except switched), and this is multiplied by the gradient on its output during the chain rule.

7 Neural Networks

It is possible to introduce neural networks without appealing to brain analogies. In the section on linear classification we computed scores for different visual categories given the image using the formula s=Wx, where W was a matrix and x was an input column vector containing all pixel data of the image. In the case of CIFAR-10, x is a [3072x1] column vector, and W is a [10x3072] matrix, so that the output scores is a vector of 10 class scores.

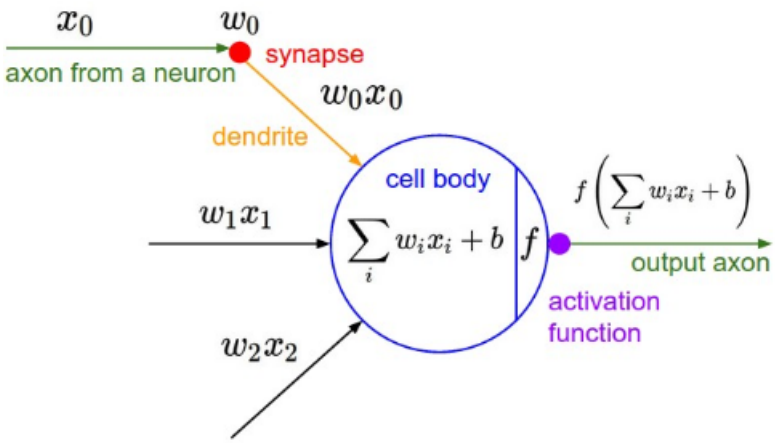
An example neural network would instead compute s=W_2max(0,W_1x). Here, W1 could be, for example, a [100x3072] matrix transforming the image into a 100-dimensional intermediate vector. The function max(0,) is a non-linearity that is applied elementwise. There are several choices we could make for the non-linearity (which well study below), but this one is a common choice and simply thresholds all activations that are below zero to zero. Finally, the matrix W_2 would then be of size [10x100], so that we again get 10 numbers out that we interpret as the class scores. Notice that the non-linearity is critical computationally - if we left it out, the two matrices could be collapsed to a single matrix, and therefore the predicted class scores would again be a linear function of the input. The non-linearity is where we get the wiggle. The parameters W_2,W_1 are learned with stochastic gradient descent, and their gradients are derived with chain rule (and computed with backpropagation).

A three-layer neural network could analogously look like $s = W_3max(0, W_2max(0, W_1x))$, where all of W_3, W_2, W_1 are parameters to be learned. The sizes of the intermediate hidden vectors are hyperparameters of the network and well see how we can set them later.

7.1 Modelling a neuron

Based on this rate code interpretation, we model the firing rate of the neuron with an activation function f, which represents the frequency of the spikes along the axon. Historically, a common choice of activation function is the sigmoid function , since it takes a real-valued input (the signal strength after the sum) and squashes

it to range between 0 and 1.



7.2 Single Neuron as a Linear Classifier

The mathematical form of the model Neurons forward computation might look familiar to you. As we saw with linear classifiers, a neuron has the capacity to like (activation near one) or dislike (activation near zero) certain linear regions of its input space. Hence, with an appropriate loss function on the neurons output, we can turn a single neuron into a linear classifier:

Binary Softmax classifier For example, we can interpret $(\sum_i w_i x_i + b)$ to be the probability of one of the classes $P(y_i = 1 x_i; w)$ The probability of the other class would be $P(y_i = 1 x_i; w) = 1 - P(y_i = 0 x_i; w)$, since they must sum to one. With this interpretation, we can formulate the cross-entropy loss as we have seen in the Linear Classification section, and optimizing it would lead to a binary Softmax classifier (also known as logistic regression). Since the sigmoid function is restricted to be between 0-1, the predictions of this classifier are based on whether the output of the neuron is greater than 0.5.

Binary SVM classifier Alternatively, we could attach a max-margin hinge loss to the output of the neuron and train it to become a binary Support Vector Machine.
Regularization interpretation The regularization loss in both SVM/Softmax cases could in this biological view be interpreted as gradual forgetting, since it would have the effect of driving all synaptic weights w towards zero after every parameter update.

7.3 Commonly used Activation Functions

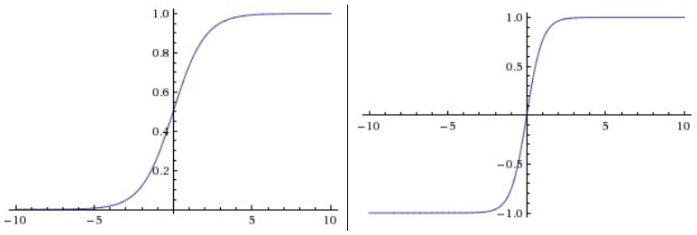
Sigmoid function

The sigmoid non-linearity has the mathematical form $(x) = \frac{e^x}{(1+e^x)}$and is shown in the image above on the left. As alluded to in the previous section, 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.In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:
1.**Sigmoids saturate and kill gradients** A very undesirable property of the sigmoid neuron is that when the neurons activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gates 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.

2.**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^Tx+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).

Tanh

It 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 non-linearity.



Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].

ReLU

The Rectified Linear Unit has become very popular in the last few years. It computes the function f(x)=max(0,x). In other words, the activation is simply threshold at zero (see image above on the left). It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form. Unfortunately, ReLU units can be fragile during training and can die.

Leaky ReLU

Leaky ReLUs are one attempt to fix the dying ReLU problem. Instead of the function being zero when x ¡ 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes f(x)=1(x¡0)(kx)+1(x¿=0)(x) where k is a small constant. Some people report success with this form of activation function, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons.

Maxout

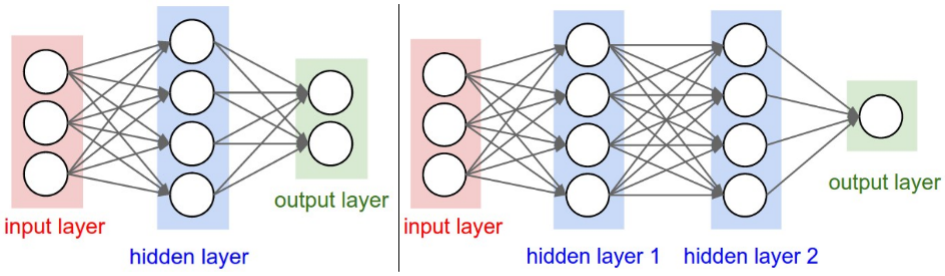
The Maxout neuron computes the function $max({w_1}^T x + b_1, {w_2}^T x + b_2)$. Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $w_1, b_1 = 0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

7.4 Neural Network Architectures

Layer-wise Organisation

Neural Networks as neurons in graphs 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 blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons. For regular neural networks, 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.



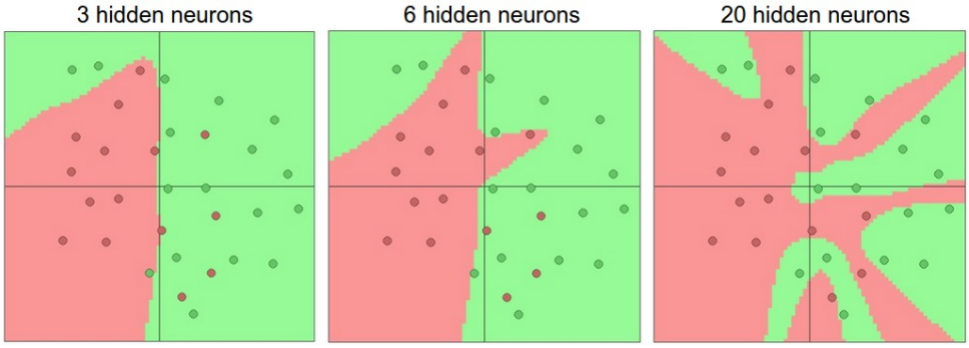
Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. **Right:** A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses)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

7.4.1 Example feed-forward computation

The forward pass of a fully-connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function.

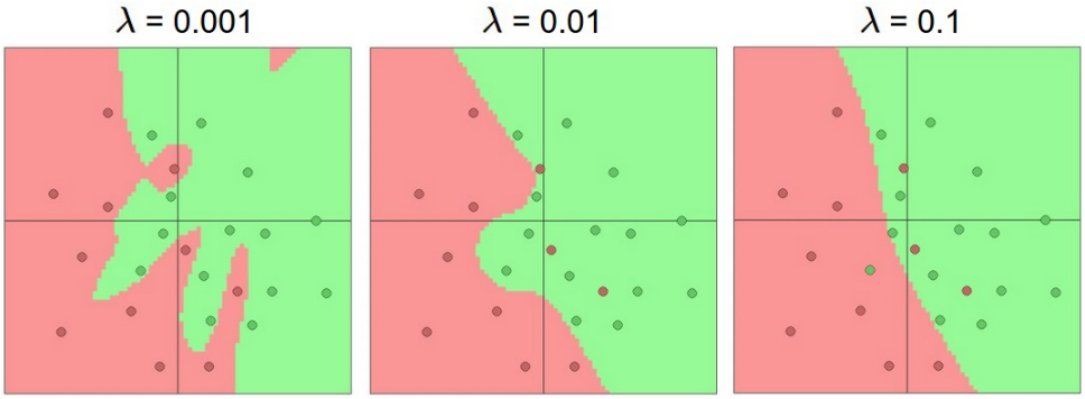
7.4.2 Setting number of layers and their sizes

How do we decide on what architecture to use when faced with a practical problem? Should we use no hidden layers? One hidden layer? Two hidden layers? How large should each layer be? First, note that as we increase the size and number of layers in a Neural Network, the capacity of the network increases. That is, the space of representable functions grows since the neurons can collaborate to express many different functions.



However, this is both a blessing (since we can learn to classify more complicated data) and a curse (since it is easier to overfit the training data). Overfitting occurs when a model with high capacity fits the noise in the data instead of the (assumed)

underlying relationship. There are many other preferred ways to prevent overfitting in Neural Networks that we will discuss later (such as L2 regularization, dropout, input noise). In practice, it is always better to use these methods to control overfitting instead of the number of neurons.
To reiterate, the regularization strength is the preferred way to control the overfitting of a neural network. We can look at the results achieved by three different settings:

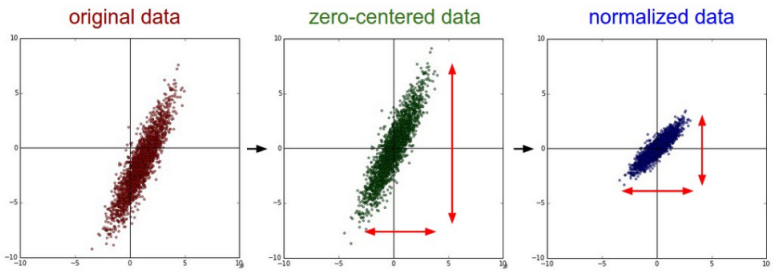


The effects of regularization strength: Each neural network above has 20 hidden neurons, but changing the regularization strength makes its final decision regions smoother with a higher regularization. You can play with these examples in this ConvNetsJS demo.

8 Setting up Neural Networks

8.1 Data Pre Processing

There are three common forms of data preprocessing a data matrix X, where we will assume that X is of size [N x D] (N is the number of data, D is their dimensionality).
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.
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. It only makes sense to apply this preprocessing if you have a reason to believe that different input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm. In case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional preprocessing step.



Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

PCA and Whitening is another form of preprocessing.

8.2 Weight Initialization

We have seen how to construct a Neural Network architecture, and how to preprocess the data. Before we can begin to train the network we have to initialize its parameters.

All zero initialization A reasonable-sounding idea then might be to set all the initial weights to zero, which we expect to be the best guess in expectation. This turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

Small random numbers Therefore, we still want the weights to be very close to zero, but as we have argued above, not identically zero. As a solution, it is common to initialize the weights of the neurons to small numbers and refer to doing so as symmetry breaking.

Its not necessarily the case that smaller numbers will work strictly better. For example, a Neural Network layer that has very small weights will during backpropagation compute very small gradients on its data (since this gradient is proportional to the value of the weights). This could greatly diminish the gradient signal flowing backward through a network, and could become a concern for deep networks.

Calibrating the variances with 1/sqrt(n) We can normalize the variance of each neurons output to 1 by scaling its weight vector by the square root of its fan-in (i.e. its number of inputs).The initialization w = np.random.randn(n) * sqrt(2.0/n) is the current recommendation for use in practice in the specific case of neural networks with ReLU neurons.

Sparse initialization Another way to address the uncalibrated variances problem is to set all weight matrices to zero, but to break symmetry every neuron is randomly connected (with weights sampled from a small gaussian as above) to a fixed number of neurons below it. A typical number of neurons to connect to may be as small as 10.

Initializing the biases It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights. For ReLU non-linearities, some people like to use small constant value such as 0.01 for all biases because this ensures that all ReLU units fire in the beginning and therefore obtain and propagate some gradient. However, it is not clear if this provides a consistent improvement (in fact some results seem to

indicate that this performs worse) and it is more common to simply use 0 bias initialization.

8.3 Regularization

There are several ways of controlling the capacity of Neural Networks to prevent overfitting:

L2 regularization

L2 regularization is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. That is, for every weight w in the network, we add the term 12w2 to the objective, where is the regularization strength. It is common to see the factor of 12 in front because then the gradient of this term with respect to the parameter w is simply w instead of 2w.

L1 regularisation

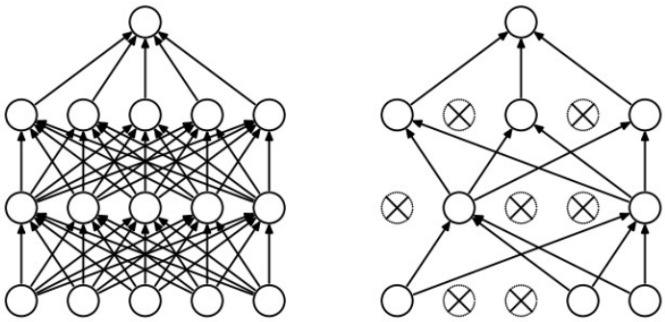
L1 regularization is another relatively common form of regularization, where for each weight w we add the term w to the objective. It is possible to combine the L1 regularization with the L2 regularization: 1w+2w2 (this is called Elastic net regularization).

Max norm constraints

Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint.One of its appealing properties is that network cannot explode even when the learning rates are set too high because the updates are always bounded.

Dropout

Dropout is an extremely effective and simple strategy. While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise.



(a) Standard Neural Net (b) After applying dropout.

Figure taken from the Dropout paper that illustrates the idea. During training, Dropout can be interpreted as sampling a Neural Network within the full Neural Network, and only updating the parameters of the sampled network based on the input data. (However, the exponential number of possible sampled networks are not independent because they share the parameters.) During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks (more about ensembles in the next section).

8.4 Loss functions

We have discussed the regularization loss part of the objective, which can be seen as penalizing some measure of complexity of the model. The second part of an objective is the data loss, which in a supervised learning problem measures the compatibility between a prediction (e.g. the class scores in classification) and the ground truth label.

Classification is the case that we have so far discussed at length. Here, we assume a dataset of examples and a single correct label (out of a fixed set) for each example. The commonly used loss functions are **SVM,squared hinge loss and Softmax classifier using cross-entropy loss**.

9 Learning and Evaluation

In the previous sections weve discussed the static parts of a Neural Networks: how we can set up the network connectivity, the data, and the loss function. This section is devoted to the dynamics, or in other words, the process of learning the parameters and finding good hyperparameters.

9.1 Checks

Gradient checks

In theory, performing a gradient check is as simple as comparing the analytic gradient to the numerical gradient.Use relative error for comparison. Use double-precision numbers.

Sanity checks

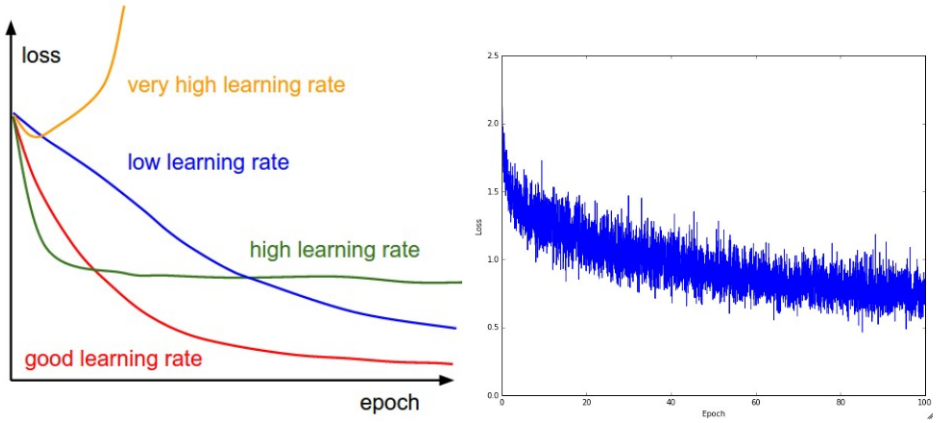
Check loss and overfit data for a small subset of data.

9.2 Learning process

There are multiple useful quantities you should monitor during training of a neural network. These plots are the window into the training process and should be utilized to get intuitions about different hyperparameter settings and how they should be changed for more efficient learning.

9.2.1 Loss function

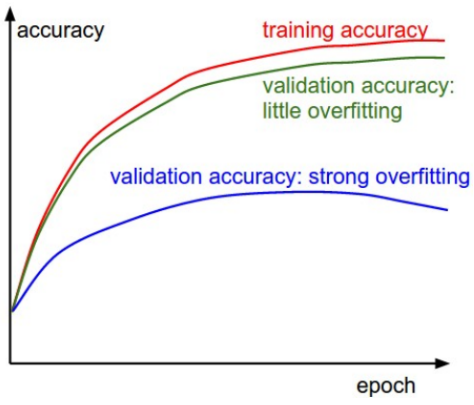
The first quantity that is useful to track during training is the loss, as it is evaluated on the individual batches during the forward pass.



Left: A cartoon depicting the effects of different learning rates. With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape. **Right:** An example of a typical loss function over time, while training a small network on CIFAR-10 dataset. This loss function looks reasonable (it might indicate a slightly too small learning rate based on its speed of decay, but it's hard to say), and also indicates that the batch size might be a little too low (since the cost is a little too noisy).

The amount of wiggle in the loss is related to the batch size. When the batch size is 1, the wiggle will be relatively high. When the batch size is the full dataset, the wiggle will be minimal because every gradient update should be improving the loss function monotonically (unless the learning rate is set too high).

9.2.2 Train/val accuracy



The gap between the training and validation accuracy indicates the amount of overfitting. Two possible cases are shown in the diagram on the left. The blue validation error curve shows very small validation accuracy compared to the training accuracy, indicating strong overfitting (note, it's possible for the validation accuracy to even start to go down after some point). When you see this in practice you probably want to increase regularization (stronger L2 weight penalty, more dropout, etc.) or collect more data. The other possible case is when the validation accuracy tracks the training accuracy fairly well. This case indicates that your model capacity is not high enough: make the model larger by increasing the number of parameters.

9.2.3 Weights:Updates ratio

The last quantity you might want to track is the ratio of the update magnitudes to the value magnitudes. Note: updates, not the raw gradients (e.g. in vanilla SGD this would be the gradient multiplied by the learning rate). You might want to evaluate and track this ratio for every set of parameters independently. A rough heuristic is that this ratio should be somewhere around 1e-3. If it is lower than this then the learning rate might be too low. If it is higher then the learning rate is likely too high.

9.3 Parameter updates

Once the analytic gradient is computed with backpropagation, the gradients are used to perform a parameter update.

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: x += - learning-rate * dx where 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.

Momentum Update

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.

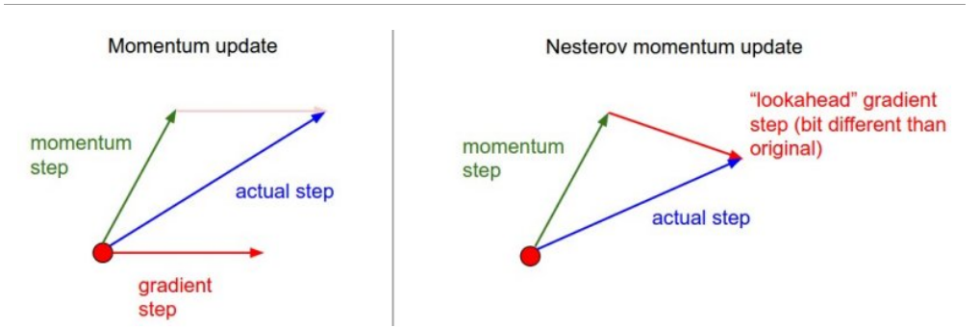
$$v = mu * v - learning - rate * dx$$

$$x+ = v$$

Here we see an introduction of a v variable that is initialized at zero, and an additional hyperparameter (mu).Its physical meaning is more consistent with the coefficient of friction. Effectively, this variable 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.

Nesterov momentum

Nesterov Momentum is a slightly different version of the momentum update that has recently been gaining popularity. It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistenly works slightly better than standard momentum.



Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

9.4 Annealing the learning rate

In training deep networks, it is usually helpful to anneal the learning rate over time. Good intuition to have in mind is that with a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function.
Step decay: Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model. One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.
Exponential decay. has the mathematical form $a = \frac{a_0}{e^(kt)}$, where a_0,k are hyperparameters and t is the iteration number (but you can also use units of epochs).
1/t decay has the mathematical form $a = \frac{a_0}{1+kt}$ where a_0, k are hyperparameters and t is the iteration number.

9.5 Per-parameter adaptive learning rates

All previous approaches weve discussed so far manipulated the learning rate globally and equally for all parameters. Tuning the learning rates is an expensive

process, so much work has gone into devising methods that can adaptively tune the learning rates, and even do so per parameter.
Adagrad cache += dx**2
x += - learning-rate * dx / (np.sqrt(cache) + eps)
Notice that 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.A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.
Other methods include **RMSprop** and **Adam**.

9.6 Hyperparameter Optimization

As weve seen, training Neural Networks can involve many hyperparameter settings. The most common hyperparameters in context of Neural Networks include: the initial learning rate, learning rate decay schedule (such as the decay constant) and regularization strength (L2 penalty, dropout strength).
Prefer one validation fold to cross-validation,use random search and ranges.

9.7 Model Ensembles

In practice, one reliable approach to improving the performance of Neural Networks by a few percent is to train multiple independent models, and at test time average their predictions. As the number of models in the ensemble increases, the performance typically monotonically improves (though with diminishing returns). Moreover, the improvements are more dramatic with higher model variety in the ensemble.Different methods include

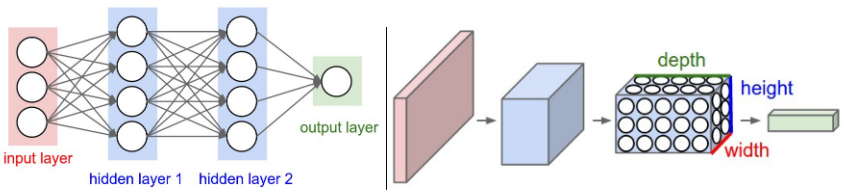
- Same model, different initializations
- Different checkpoints of a single model
- Running average of parameters during training
- Top models discovered during cross-validation

10 Convolutional Neural Networks

Convolutional Neural Networks are very similar to ordinary Neural Networks from the previous chapter: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

10.1 Architecture Overview

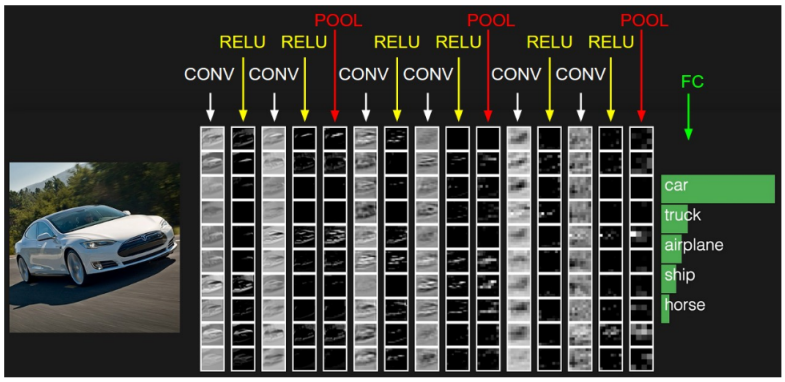
In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.)



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

10.2 ConvNets Layers

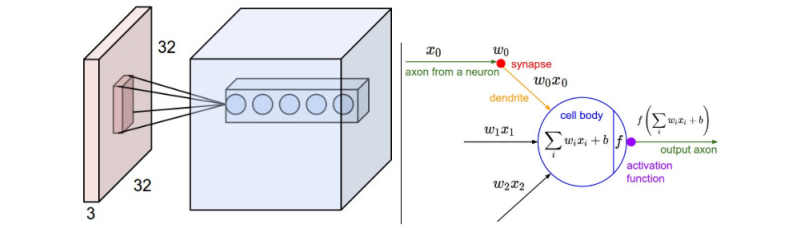
Every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture.



The activations of an example ConvNet architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one. The full web-based demo is shown in the header of our website. The architecture shown here is a tiny VGG Net, which we will discuss later.

10.2.1 Convulation Layer

The CONV layers 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.



Left: 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). Note, 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. **Right:** The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

Three hyperparameters control the size of the output volume: the **depth, stride and zero-padding**.
First, the **depth** of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input.
Second, 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.
Sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly as well see soon we will use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

Parameter sharing

Parameter sharing scheme is used in Convolutional Layers to control the number of parameters. Using the real-world example above, we see that there are 55*55*96 = 290,400 neurons in the first Conv Layer, and each has 11*11*3 = 363 weights and 1 bias. Together, this adds up to 290400 * 364 = 105,705,600 parameters on the first layer of the ConvNet alone. Clearly, this number is very high. It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: That if one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x2,y2). In other words, denoting a single 2-dimensional slice of depth as a depth slice (e.g. a volume of size [55x55x96] has 96 depth slices, each of size [55x55]), we are going to constrain the neurons in each depth slice to use the same weights and

bias. With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of 96*11*11*3 = 34,848 unique weights, or 34,944 parameters (+96 biases).

Summary

Accepts a volume of size $\boldsymbol{W_1 H_1 D_1}$
Requires four hyperparameters: Number of filters $\boldsymbol{K}$, their spatial extent $\boldsymbol{F}$, the stride $\boldsymbol{S}$, the amount of zero padding $\boldsymbol{P}$.
Produces a volume of size $\boldsymbol{W_2 H_2 D_2}$ where:

$$\boldsymbol{W_2 = \frac{W_1 F + 2P}{S} + 1}$$

$\boldsymbol{H_2 = \frac{H_1 F + 2P}{S} + 1}$(i.e. width and height are computed equally by symmetry)

$$\boldsymbol{D_2 = K}$$

With parameter sharing, it introduces $\boldsymbol{FxFxD_1}$ weights per filter, for a total of $\boldsymbol{(FFD1)K}$ weights and$\boldsymbol{K}$ biases.
In the output volume, the d-th depth slice (of size $\boldsymbol{W_2 H_2}$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of $\boldsymbol{S}$, and then offset by d-th bias.

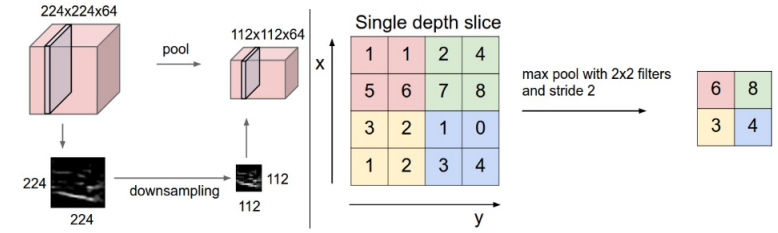
10.2.2 Pooling Layer

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, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 3/4th of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice).
Accepts a volume of size $\boldsymbol{W_1 H_1 D_1}$
Requires two hyperparameters: their spatial extent $\boldsymbol{F}$, the stride $\boldsymbol{S}$,
Produces a volume of size $\boldsymbol{W_2 H_2 D_2}$ where:

$$\boldsymbol{W_2 = \frac{W_1 F}{S} + 1}$$

$$H_2 = \frac{H_1 F}{S} + 1$$
$$D_2 = D_1$$

Introduces **zero parameters** since it computes a fixed function of the input. For Pooling layers, it is not common to pad the input using zero-padding.



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

10.2.3 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.

10.3 Layer Patterns

he 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.
INPUT - [[CONV - RELU]*N - POOL]*M - [FC - RELU]*K - FC

10.4 Embedding the codes with t-SNE

ConvNets can be interpreted as gradually transforming the images into a representation in which the classes are separable by a linear classifier. We can get a rough idea about the topology of this space by embedding images into two dimensions so that their low-dimensional representation has approximately equal distances than their high-dimensional representation. There are many embedding methods that

have been developed with the intuition of embedding high-dimensional vectors in a low-dimensional space while preserving the pairwise distances of the points. Among these, t-SNE is one of the best-known methods that consistently produces visually-pleasing results.

References

- CS 231N Convolutional Neural Networks for Visual Recognition by Stanford University. The course instructors were Prof. Fei-Fei Li, Justin Johnson and Serena Yeung.
- Support Vector Machine (SVM) for Multi-Class Pattern Recognition by J. Weston and C.Watkins
- ImageNet Classification with Deep Convolutional Neural Networks by Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton for comparing ReLU vs tanh
- Understanding the difficulty of training deep feedforward neural networks by Xavier Glorot for a good initialisation.
- Dropout: A Simple Way to Prevent Neural Networks from Overfitting by Nitish Srivastava
- LeNet Gradient Based Learning for Document Recognition by Yann LeCun
- AlexNet ImageNet Classification with Deep Convolutional Neural Networks by Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton
- ZFNet Visualizing and Understanding Convolutional Networks by Matthew D Zeiler, Rob Fergus
- GoogleNet Going Deeper with Convolutions by Szegedy et al.
- VGGNet Very Deep Convolutional Networks for Large-Scale Visual Recognition by Karen Simonyan and Andrew Zisserman.
- ResNet Deep Residual Learning for Image Recognition by Kaiming He