Table of Contents

CS231n Convolutional Neural Networks for Visual Recognition

Course Website

These notes accompany the Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition. For questions/concerns/bug reports, please submit a pull request directly to our git repo.

Spring 2023 Assignments

Assignment #1: Image Classification, kNN, SVM, Softmax, Fully Connected Neural Network

Assignment #2: Fully Connected and Convolutional Nets, Batch Normalization, Dropout, Pytorch & Network Visualization

Assignment #3: Network Visualization, Image Captioning with RNNs and Transformers, Generative Adversarial Networks, Self-Supervised Contrastive Learning

Module 0: Preparation

Software Setup

Python / Numpy Tutorial (with Jupyter and Colab)

Module 1: Neural Networks

Image Classification: Data-driven Approach, k-Nearest Neighbor, train/val/test splits

L1/L2 distances, hyperparameter search, cross-validation

Linear classification: Support Vector Machine, Softmax

parameteric approach, bias trick, hinge loss, cross-entropy loss, L2 regularization, web demo

Optimization: Stochastic Gradient Descent

optimization landscapes, local search, learning rate, analytic/numerical gradient

Backpropagation, Intuitions

chain rule interpretation, real-valued circuits, patterns in gradient flow

Neural Networks Part 1: Setting up the Architecture

model of a biological neuron, activation functions, neural net architecture, representational power

Neural Networks Part 2: Setting up the Data and the Loss

preprocessing, weight initialization, batch normalization, regularization (L2/dropout), loss functions

Neural Networks Part 3: Learning and Evaluation

gradient checks, sanity checks, babysitting the learning process, momentum (+nesterov), second-order methods, Adagrad/RMSprop, hyperparameter optimization, model ensembles

Putting it together: Minimal Neural Network Case Study

minimal 2D toy data example

Module 2: Convolutional Neural Networks

Convolutional Neural Networks: Architectures, Convolution / Pooling Layers

layers, spatial arrangement, layer patterns, layer sizing patterns, AlexNet/ZFNet/VGGNet case studies, computational considerations

Understanding and Visualizing Convolutional Neural Networks

tSNE embeddings, deconvnets, data gradients, fooling ConvNets, human comparisons

Transfer Learning and Fine-tuning Convolutional Neural Networks

Student-Contributed Posts

Taking a Course Project to Publication

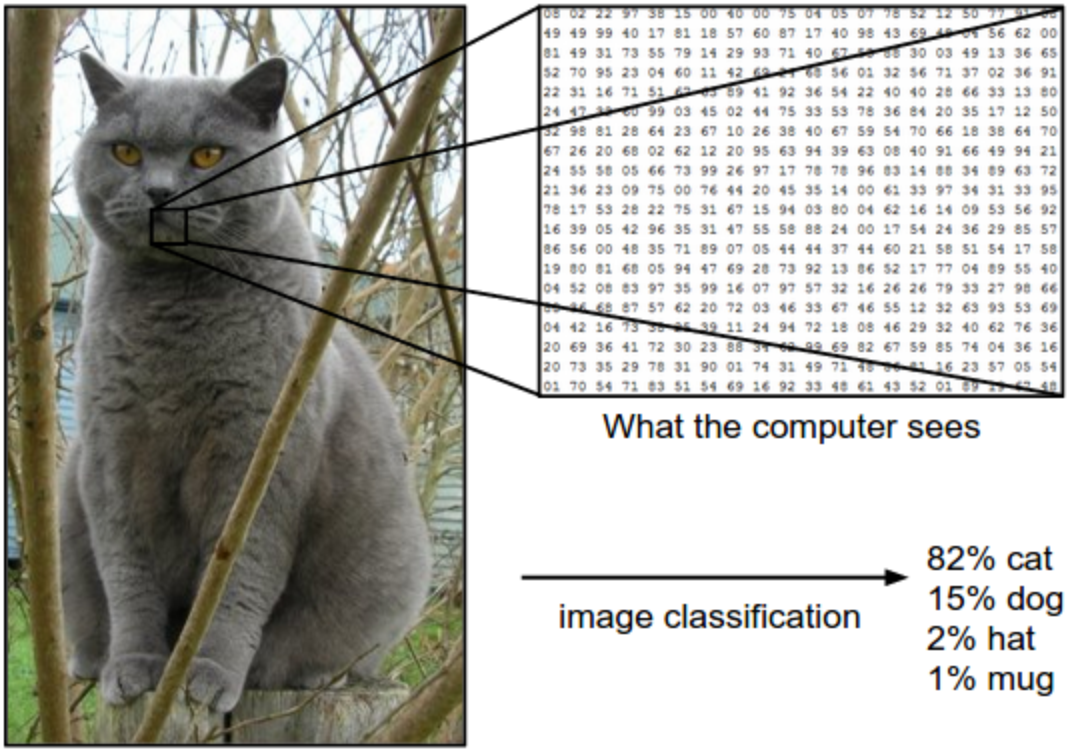
Recurrent Neural Networks

This is an introductory lecture designed to introduce people from outside of Computer Vision to the Image Classification problem, and the data-driven approach. The Table of Contents:

Image Classification

Motivation. In this section 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. Moreover, as we will see later in the course, many other seemingly distinct Computer Vision tasks (such as object detection, segmentation) can be reduced to image classification.

Example. 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 this quarter of a million numbers into a single label, such as *"cat"*.

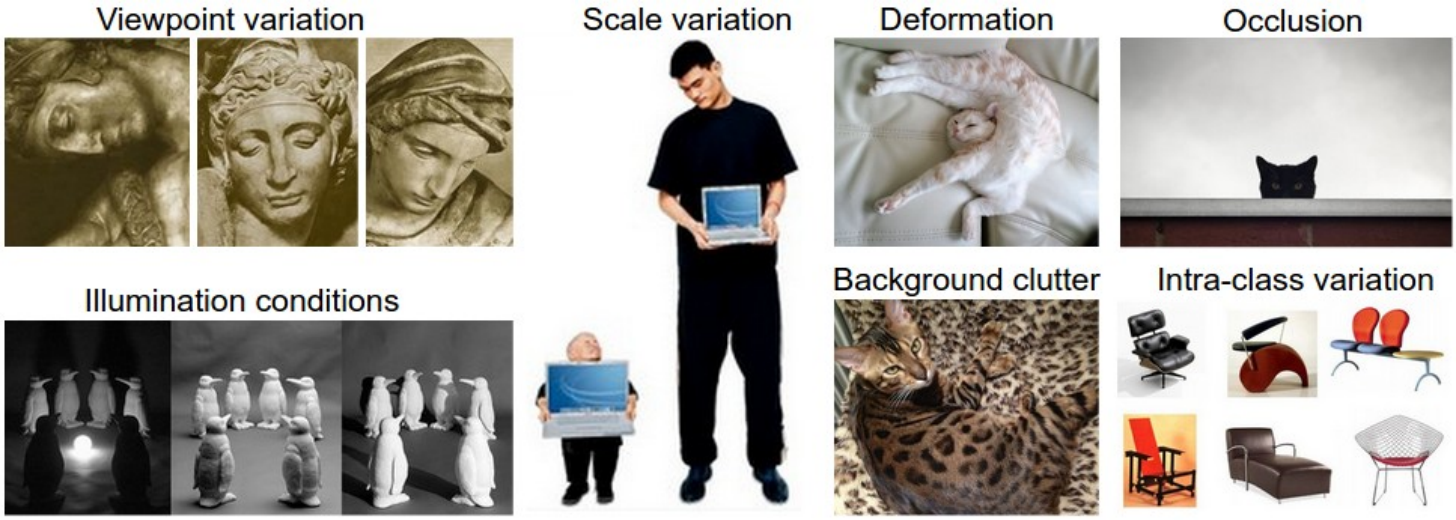


The task in Image Classification is to predict a single label (or a distribution over labels as shown here to indicate our confidence) for a given image. Images are 3-dimensional arrays of integers from 0 to 255, of size Width x Height x 3. The 3 represents the three color channels Red, Green, Blue.

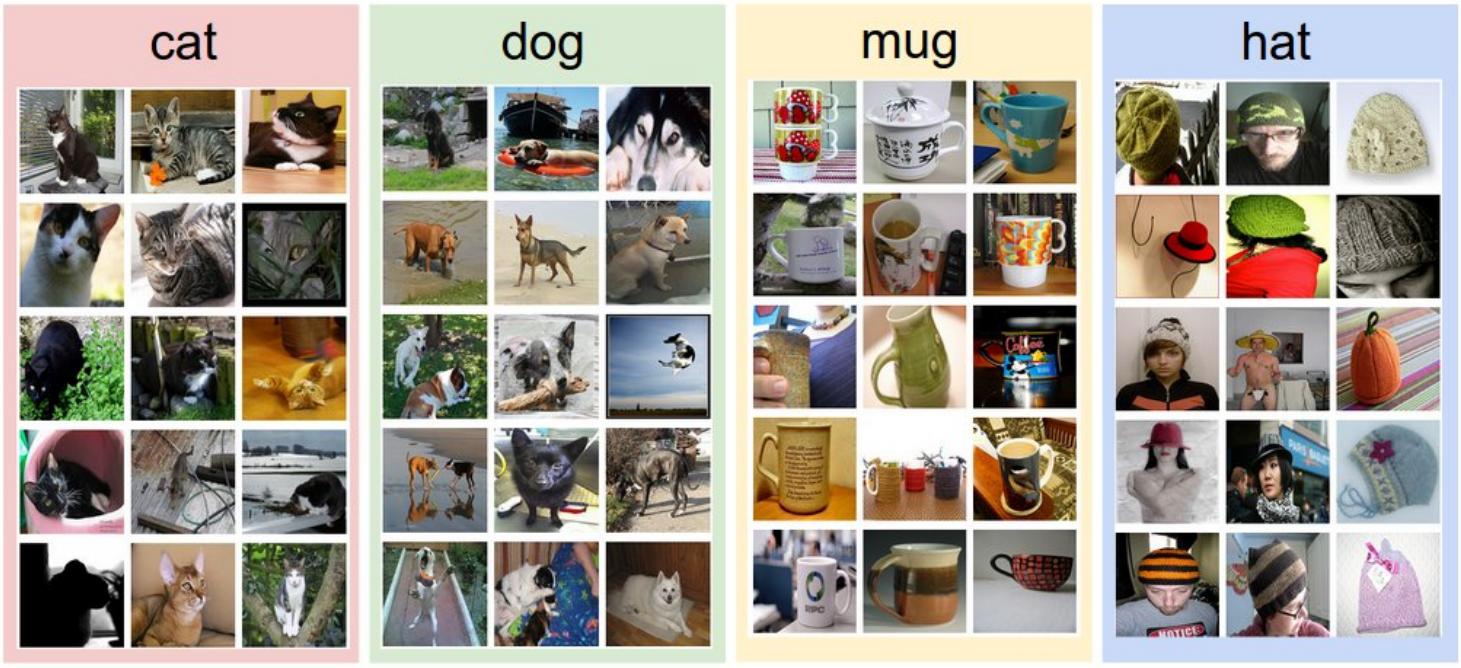
Challenges. Since this task of recognizing a visual concept (e.g. cat) is relatively trivial for a human to perform, it is worth considering the challenges involved from the perspective of a Computer Vision algorithm. As we present (an inexhaustive) list of challenges below, keep in mind the raw representation of images as a 3-D array of brightness values:

- **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.



Data-driven approach. How might we go about writing an algorithm that can classify images into distinct categories? Unlike writing an algorithm for, for example, sorting a list of numbers, it is not obvious how one might write an algorithm for identifying cats in images. Therefore, instead of trying to specify what every one of the categories of interest look like directly in code, the approach that we will take is not unlike one you would take with a child: we're going to provide the computer with many examples of each class and then develop learning algorithms that look at these examples and learn about the visual appearance of each class. This approach is referred to as a *data-driven approach*, since it relies on first accumulating a *training dataset* of labeled images. Here is an example of what such a dataset might look like:



An example training set for four visual categories. In practice we may have thousands of categories and hundreds of thousands of images for each category.

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

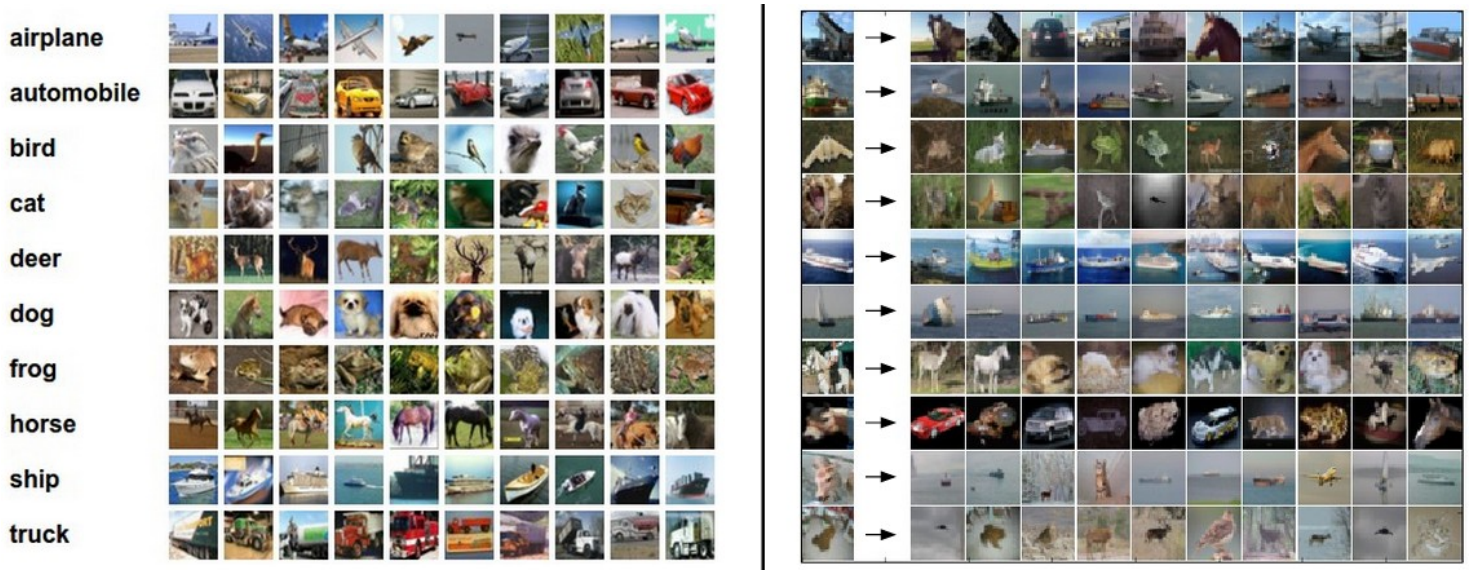
follows:

- **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 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*).

Nearest Neighbor Classifier

As our first approach, we will develop what we call a **Nearest Neighbor Classifier**. 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.

Example image classification dataset: CIFAR-10. One popular toy image classification dataset is the CIFAR-10 dataset. This dataset consists of 60,000 tiny images that are 32 pixels high and wide. Each image is labeled with one of 10 classes (for example *"airplane, automobile, bird, etc"*). These 60,000 images are partitioned into a training set of 50,000 images and a test set of 10,000 images. In the image below you can see 10 random example images from each one of the 10 classes:



Left: Example images from the CIFAR-10 dataset. Right: first column shows a few test images and next to each we show the top 10 nearest neighbors in the training set according to pixel-wise difference.

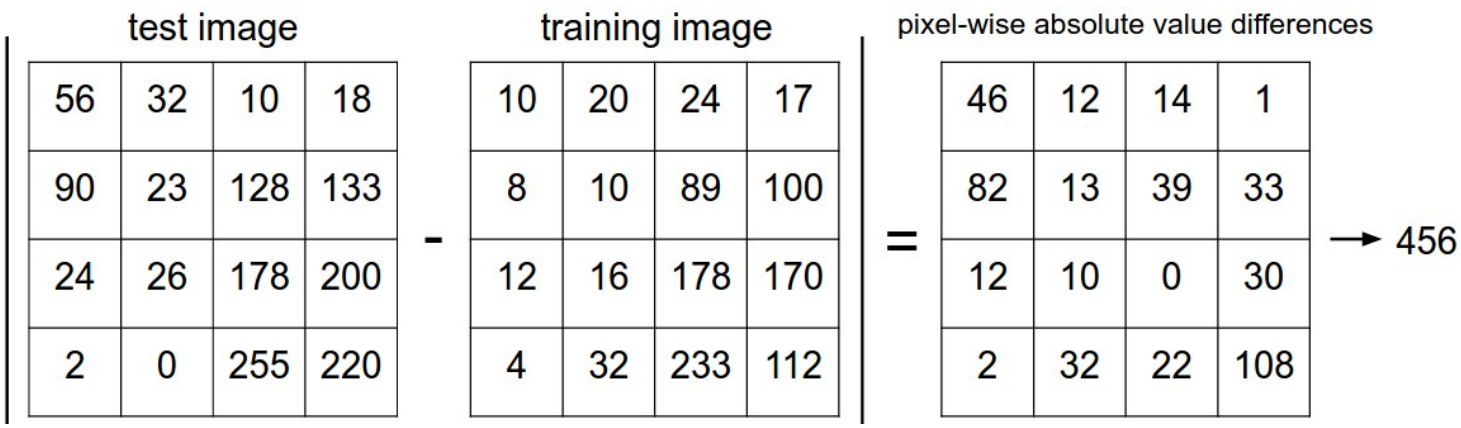
Suppose now that we are given the CIFAR-10 training set of 50,000 images (5,000 images for every one of the labels), and we wish to label the remaining 10,000. The nearest neighbor classifier will take a test image, compare it to every single one of the training images, and predict the label of the closest training image. In the image above and on the right you can see an example result of such a procedure for 10 example test

images. Notice that in only about 3 out of 10 examples an image of the same class is retrieved, while in the other 7 examples this is not the case. For example, in the 8th row the nearest training image to the horse head is a red car, presumably due to the strong black background. As a result, this image of a horse would in this case be mislabeled as a car.

You may have noticed that we left unspecified the details of exactly how we compare two images, which in this case are just two blocks of 32 x 32 x 3. 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 I_1, I_2 , a reasonable choice for comparing them might be the **L1 distance**:

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

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.

Let's also look at how we might implement the classifier in code. First, let's load the CIFAR-10 data into memory as 4 arrays: the training data/labels and the test data/labels. In the code below, `Xtr` (of size 50,000 x 32 x 32 x 3) holds all the images in the training set, and a corresponding 1-dimensional array `Ytr` (of length 50,000) holds the training labels (from 0 to 9):

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

Now that we have all images stretched out as rows, here is how we could train and evaluate a classifier:

```
nn = NearestNeighbor() # create a Nearest Neighbor classifier class
nn.train(Xtr_rows, Ytr) # train the classifier on the training images and labels
Yte_predict = nn.predict(Xte_rows) # predict labels on the test images
```

```
# and now print the classification accuracy, which is the average number
# of examples that are correctly predicted (i.e. label matches)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )
```

Notice that as an evaluation criterion, it is common to use the **accuracy**, which measures the fraction of predictions that were correct. Notice that all classifiers we will build satisfy this one common API: they have a `train(X,y)` function that takes the data and the labels to learn from. Internally, the class should build some kind of model of the labels and how they can be predicted from the data. And then there is a `predict(X)` function, which takes new data and predicts the labels. Of course, we've left out the meat of things - the actual classifier itself. Here is an implementation of a simple Nearest Neighbor classifier with the L1 distance that satisfies this template:

```
import numpy as np

class NearestNeighbor(object):
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

    # loop over all test rows
    for i in range(num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
      min_index = np.argmin(distances) # get the index with smallest distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

If you ran this code, you would see that this classifier only achieves **38.6%** on CIFAR-10. That's more impressive than guessing at random (which would give 10% accuracy since there are 10 classes), but nowhere near human performance (which is estimated at about 94%) or near state-of-the-art Convolutional Neural Networks that achieve about 95%, matching human accuracy (see the leaderboard of a recent Kaggle competition on CIFAR-10).

The choice of distance. There 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:

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

In other words we would be computing the pixelwise difference as before, but this time we square all of them, add them up and finally take the square root. In numpy, using the code from above we would need to only replace a single line of code. The line that computes the distances:

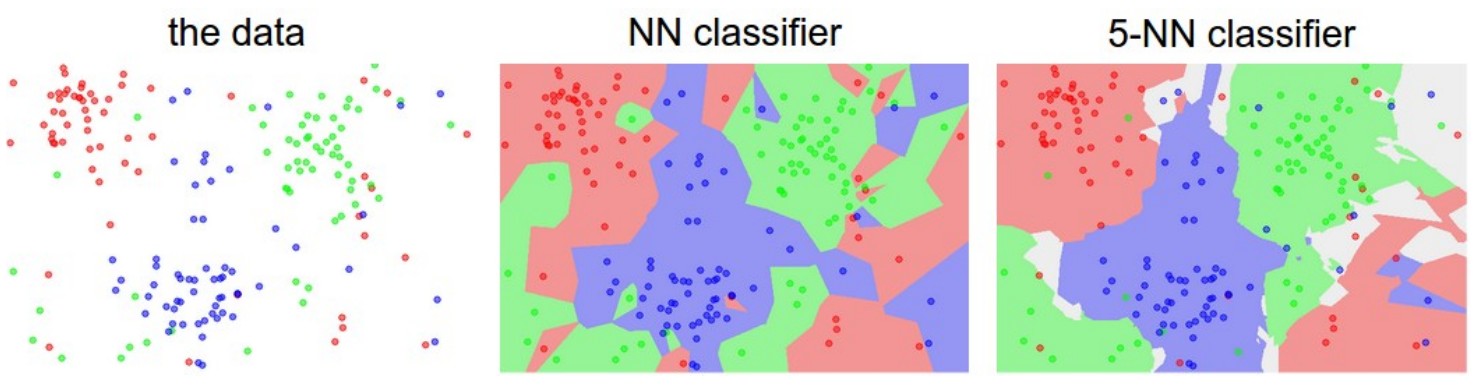
```
distances = np.sqrt(np.sum(np.square(self.Xtr - X[i,:]), axis = 1))
```

Note that I included the `np.sqrt` call above, but in a practical nearest neighbor application we could leave out the square root operation because square root is a *monotonic function*. That is, it scales the absolute sizes of the distances but it preserves the ordering, so the nearest neighbors with or without it are identical. If you ran the Nearest Neighbor classifier on CIFAR-10 with this distance, you would obtain **35.4%** accuracy (slightly lower than our L1 distance result).

L1 vs. L2. It is interesting to consider differences between the two metrics. In particular, the L2 distance is much more unforgiving than the L1 distance when it comes to differences between two vectors. That is, the L2 distance prefers many medium disagreements to one big one. L1 and L2 distances (or equivalently the L1/L2 norms of the differences between a pair of images) are the most commonly used special cases of a p-norm.

k - Nearest Neighbor Classifier

You may have noticed that 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 what's 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).

In practice, you will almost always want to use k-Nearest Neighbor. But what value of *k* should you use? We turn to this problem next.

Validation sets for Hyperparameter tuning

The k-nearest neighbor classifier requires a setting for *k*. But what number works best? Additionally, we saw 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 (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.

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 that's 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 you're 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 hyperparameters 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. Another way of looking at it is that if you tune your hyperparameters on the test set, you are effectively using the test set as the training set, and therefore the performance you achieve on it will be too optimistic with respect to what you might actually observe when you deploy your model. But if you only use the test set once at end, it remains a good proxy for measuring the **generalization** of your classifier (we will see much more discussion surrounding generalization later in the class).

> *Evaluate on the test set only a single time, at the very end.*

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**. Using CIFAR-10 as an example, we could for example use 49,000 of the training images for training, and leave 1,000 aside for validation. This validation set is essentially used as a fake test set to tune the hyper-parameters.

Here is what this might look like in the case of CIFAR-10:

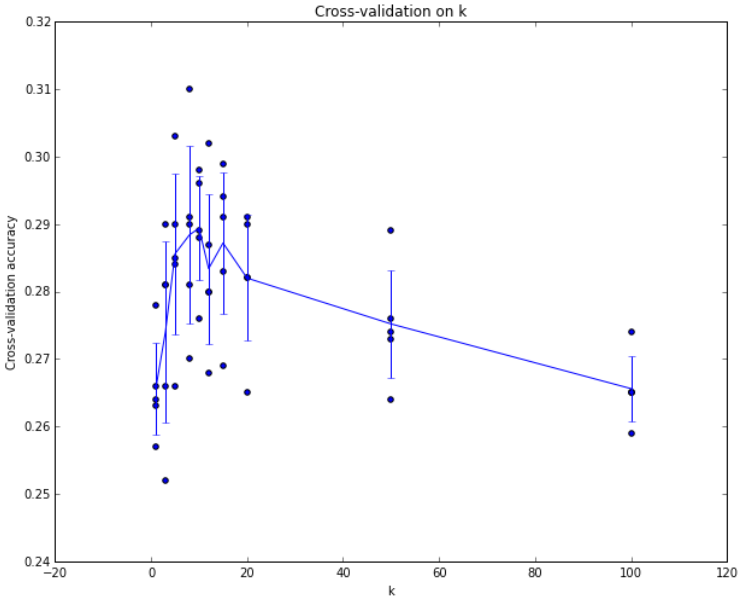
```
# assume we have Xtr_rows, Ytr, Xte_rows, Yte as before
# recall Xtr_rows is 50,000 x 3072 matrix
Xval_rows = Xtr_rows[:1000, :] # take first 1000 for validation
Yval = Ytr[:1000]
Xtr_rows = Xtr_rows[1000:, :] # keep last 49,000 for train
Ytr = Ytr[1000:]

# find hyperparameters that work best on the validation set
validation_accuracies = []
for k in [1, 3, 5, 10, 20, 50, 100]:

  # use a particular value of k and evaluation on validation data
  nn = NearestNeighbor()
  nn.train(Xtr_rows, Ytr)
  # here we assume a modified NearestNeighbor class that can take a k as input
  Yval_predict = nn.predict(Xval_rows, k = k)
  acc = np.mean(Yval_predict == Yval)
  print 'accuracy: %f' % (acc,)

  # keep track of what works on the validation set
  validation_accuracies.append((k, acc))
```

By the end of this procedure, we could plot a graph that shows which values of *k* work best. We would then stick with this value and evaluate once on the actual test set.

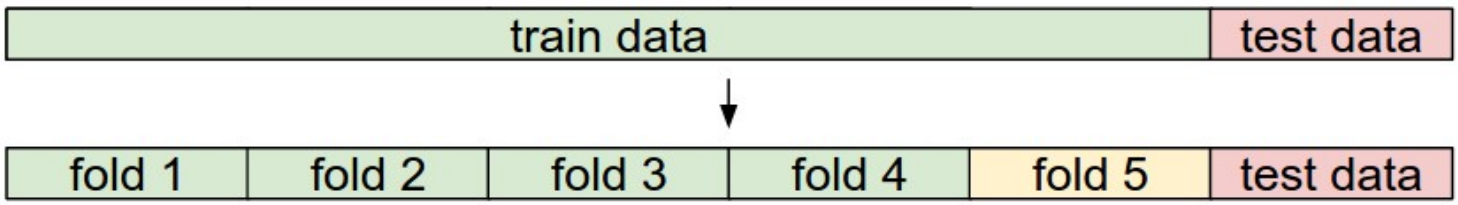
> *Split your training set into training set and a validation set. Use validation set to tune all hyperparameters. At the end run a single time on the test set and report performance.*

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.

Example of a 5-fold cross-validation run for the parameter **k**. For each value of **k** we train on 4 folds and evaluate on the 5th. Hence, for each **k** we receive 5 accuracies on the validation fold (accuracy is the y-axis, each result is a point). The trend line is drawn through the average of the results for each **k** and the error bars indicate the standard deviation. Note that in this particular case, the cross-validation suggests that a value of about **k** = 7 works best on this particular dataset (corresponding to the peak in the plot). If we used more than 5 folds, we might expect to see a smoother (i.e. less noisy) curve.



In practice. In practice, people prefer to avoid cross-validation in favor of having a single validation split, since cross-validation can be computationally expensive. The splits people tend to use is between 50%-90% of the training data for training and rest for validation. However, this depends on multiple factors: For example if the number of hyperparameters is large you may prefer to use bigger validation splits. If the number of examples in the validation set is small (perhaps only a few hundred or so), it is safer to use cross-validation. Typical number of folds you can see in practice would be 3-fold, 5-fold or 10-fold cross-validation.



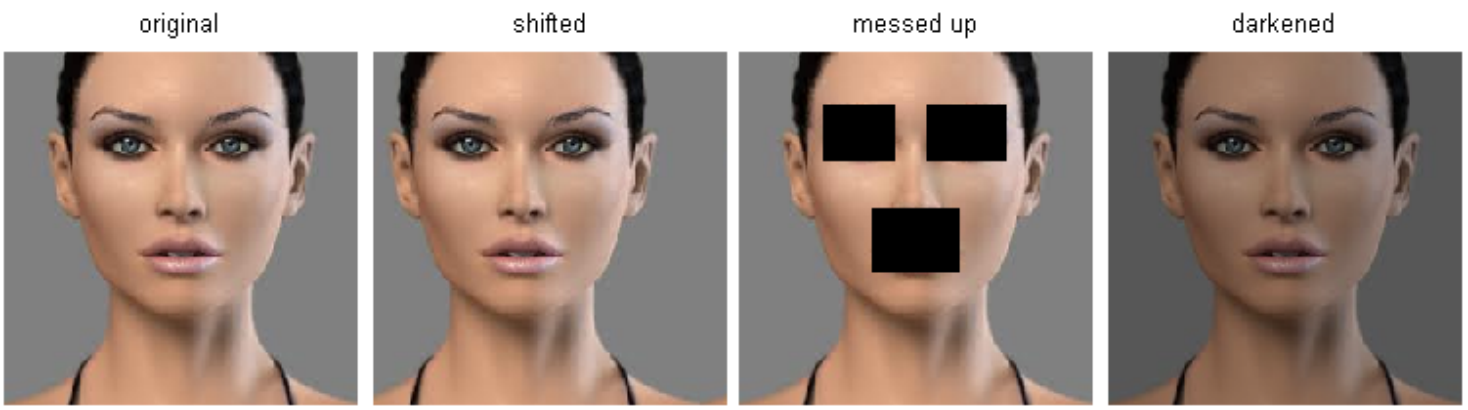
Common data splits. A training and test set is given. The training set is split into folds (for example 5 folds here). The folds 1-4 become the training set. One fold (e.g. fold 5 here in yellow) is denoted as the Validation fold and is used to tune the hyperparameters. Cross-validation goes a step further and iterates over the choice of which fold is the validation fold, separately from 1-5. This would be referred to as 5-fold cross-validation. In the very end once the model is trained and all the best hyperparameters were determined, the model is evaluated a single time on the test data (red).

Pros and Cons of Nearest Neighbor classifier.

It is worth considering some advantages and drawbacks of the Nearest Neighbor classifier. Clearly, one advantage is that it is very simple to implement and understand. Additionally, the classifier takes no time to train, since all that is required is to store and possibly index the training data. However, we pay that computational cost at test time, since classifying a test example requires a comparison to every single training example. This is backwards, since in practice we often care about the test time efficiency much more than the efficiency at training time. In fact, the deep neural networks we will develop later in this class shift this tradeoff to the other extreme: They are very expensive to train, but once the training is finished it is very cheap to classify a new test example. This mode of operation is much more desirable in practice.

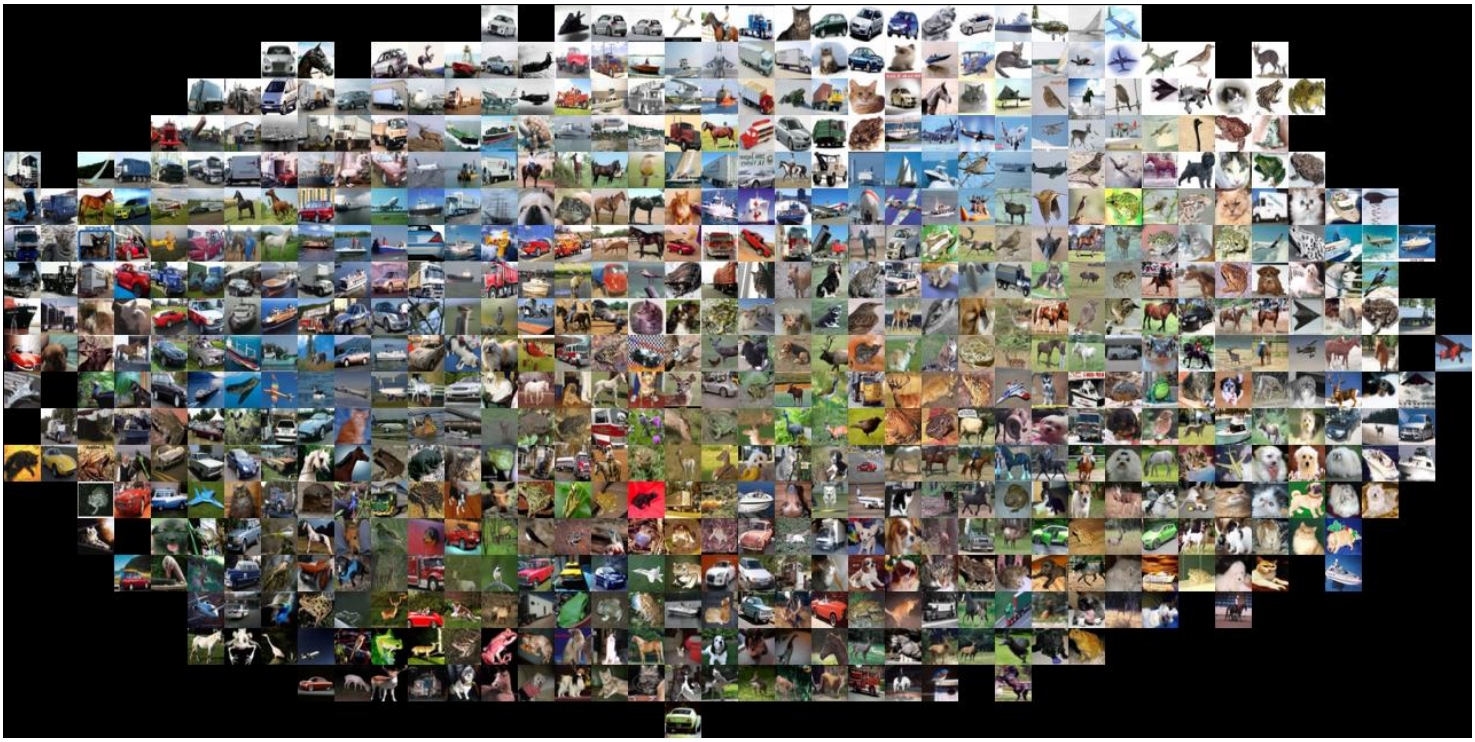
As an aside, the computational complexity of the Nearest Neighbor classifier is an active area of research, and several **Approximate Nearest Neighbor** (ANN) algorithms and libraries exist that can accelerate the nearest neighbor lookup in a dataset (e.g. FLANN). These algorithms allow one to trade off the correctness of the nearest neighbor retrieval with its space/time complexity during retrieval, and usually rely on a pre-processing/indexing stage that involves building a kdtree, or running the k-means algorithm.

The Nearest Neighbor Classifier may sometimes be a good choice in some settings (especially if the data is low-dimensional), but it is rarely appropriate for use in practical image classification settings. One problem is that images are high-dimensional objects (i.e. they often contain many pixels), and distances over high-dimensional spaces can be very counter-intuitive. The image below illustrates the point that the pixel-based L2 similarities we developed above are very different from perceptual similarities:



Pixel-based distances on high-dimensional data (and images especially) can be very unintuitive. An original image (left) and three other images next to it that are all equally far away from it based on L2 pixel distance. Clearly, the pixel-wise distance does not correspond at all to perceptual or semantic similarity.

Here is one more visualization to convince you that using pixel differences to compare images is inadequate. We can use a visualization technique called t-SNE to take the CIFAR-10 images and embed them in two dimensions so that their (local) pairwise distances are best preserved. In this visualization, images that are shown nearby are considered to be very near according to the L2 pixelwise distance we developed above:



CIFAR-10 images embedded in two dimensions with t-SNE. Images that are nearby on this image are considered to be close based on the L2 pixel distance. Notice the strong effect of background rather than semantic class differences. Click here for a bigger version of this visualization.

In particular, note that 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 dog can be seen very near a frog since both happen to be on white background. Ideally we would like images of all of the 10 classes to form their own clusters, so that images of the same class are nearby to each other regardless of irrelevant characteristics and variations (such as the background). However, to get this property we will have to go beyond raw pixels.

Summary

In summary:

- We introduced the problem of **Image Classification**, in which we are given a set of images that are all labeled with a single category. We are then asked to predict these categories for a novel set of test images and measure the accuracy of the predictions.
- We introduced a simple classifier called the **Nearest Neighbor classifier**. We saw that there are multiple hyper-parameters (such as value of k, or the type of distance used to compare examples) that are associated with this classifier and that there was no obvious way of choosing them.
- We saw that the correct way to set these hyperparameters is to split your training data into two: a training set and a fake test set, which we call **validation set**. We try different hyperparameter values and keep the values that lead to the best performance on the validation set.
- If the lack of training data is a concern, we discussed a procedure called **cross-validation**, which can help reduce noise in estimating which hyperparameters work best.

- Once the best hyperparameters are found, we fix them and perform a single **evaluation** on the actual test set.
- We saw that Nearest Neighbor can get us about 40% accuracy on CIFAR-10. It is simple to implement but requires us to store the entire training set and it is expensive to evaluate on a test image.
- Finally, we saw that the use of L1 or L2 distances on raw pixel values is not adequate since the distances correlate more strongly with backgrounds and color distributions of images than with their semantic content.

In next lectures we will embark on addressing these challenges and eventually arrive at solutions that give 90% accuracies, allow us to completely discard the training set once learning is complete, and they will allow us to evaluate a test image in less than a millisecond.

Summary: Applying kNN in practice

If you wish to apply kNN in practice (hopefully not on images, or perhaps as only a baseline) proceed as follows:

1. Preprocess your data: Normalize the features in your data (e.g. one pixel in images) to have zero mean and unit variance. We will cover this in more detail in later sections, and chose not to cover data normalization in this section because pixels in images are usually homogeneous and do not exhibit widely different distributions, alleviating the need for data normalization.
2. If your data is very high-dimensional, consider using a dimensionality reduction technique such as PCA (wiki ref, CS229ref, blog ref), NCA (wiki ref, blog ref), or even Random Projections.
3. Split your training data randomly into train/val splits. As a rule of thumb, between 70-90% of your data usually goes to the train split. This setting depends on how many hyperparameters you have and how much of an influence you expect them to have. If there are many hyperparameters to estimate, you should err on the side of having larger validation set to estimate them effectively. If you are concerned about the size of your validation data, it is best to split the training data into folds and perform cross-validation. If you can afford the computational budget it is always safer to go with cross-validation (the more folds the better, but more expensive).
4. Train and evaluate the kNN classifier on the validation data (for all folds, if doing cross-validation) for many choices of **k** (e.g. the more the better) and across different distance types (L1 and L2 are good candidates)
5. If your kNN classifier is running too long, consider using an Approximate Nearest Neighbor library (e.g. FLANN) to accelerate the retrieval (at cost of some accuracy).
6. Take note of the hyperparameters that gave the best results. There is a question of whether you should use the full training set with the best hyperparameters, since the optimal hyperparameters might change if you were to fold the validation data into your training set (since the size of the data would be larger). In practice it is cleaner to not use the validation data in the final classifier and consider it to be *burned* on estimating the hyperparameters. Evaluate the best model on the test set. Report the test set accuracy and declare the result to be the performance of the kNN classifier on your data.

Further Reading

Here are some (optional) links you may find interesting for further reading:

- A Few Useful Things to Know about Machine Learning, where especially section 6 is related but the whole paper is a warmly recommended reading.
- Recognizing and Learning Object Categories, a short course of object categorization at ICCV 2005.

Table of Contents:

Linear Classification

In the last section we introduced the problem of Image Classification, which is the task of assigning a single label to an image from a fixed set of categories. Moreover, we described the k-Nearest Neighbor (kNN) classifier which labels images by comparing them to (annotated) images from the training set. As we saw, kNN has a number of 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.

Overview. We are now going to develop a more powerful approach to image classification that we will eventually naturally 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 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.

Parameterized mapping from images to label scores

The first component of this approach is to define the score function that maps the pixel values of an image to confidence scores for each class. We will develop the approach with a concrete example. As before, let's assume a training dataset of images $x_i \in R^D$, each associated with a label y_i. Here $i = 1 \ldots N$ and $y_i \in 1 \ldots K$. That is, we have **N** examples (each with a dimensionality **D**) and **K** distinct categories. For example, in CIFAR-10 we have a training set of **N** = 50,000 images, each with **D** = 32 x 32 x 3 = 3072 pixels, and **K** = 10, since there are 10 distinct classes (dog, cat, car, etc). We will now define the score function $f: R^D \mapsto R^K$ that maps the raw image pixels to class scores.

Linear classifier. In this module we will start out with arguably the simplest possible function, a linear mapping:

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

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. However, you will often hear people use the terms *weights* and *parameters* interchangeably.

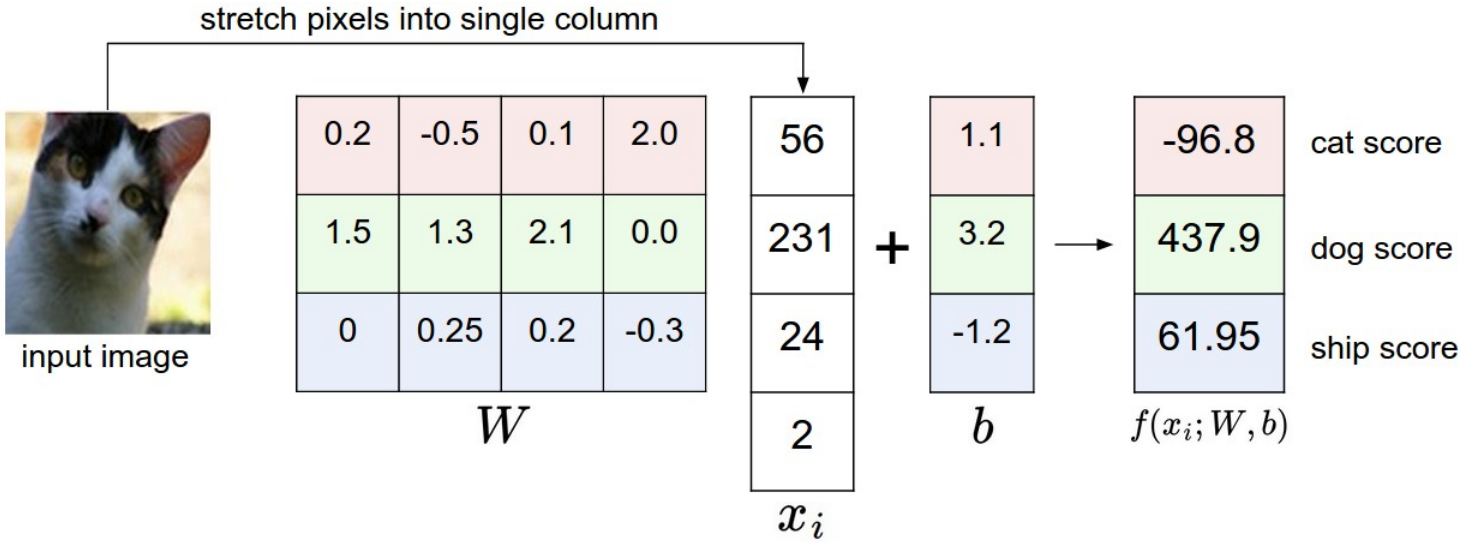
There are a few things to note:

- 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**.
- Notice also that 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 will go into much more detail about how this is done, but intuitively 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.

> *Foreshadowing: 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.*

Interpreting a linear classifier

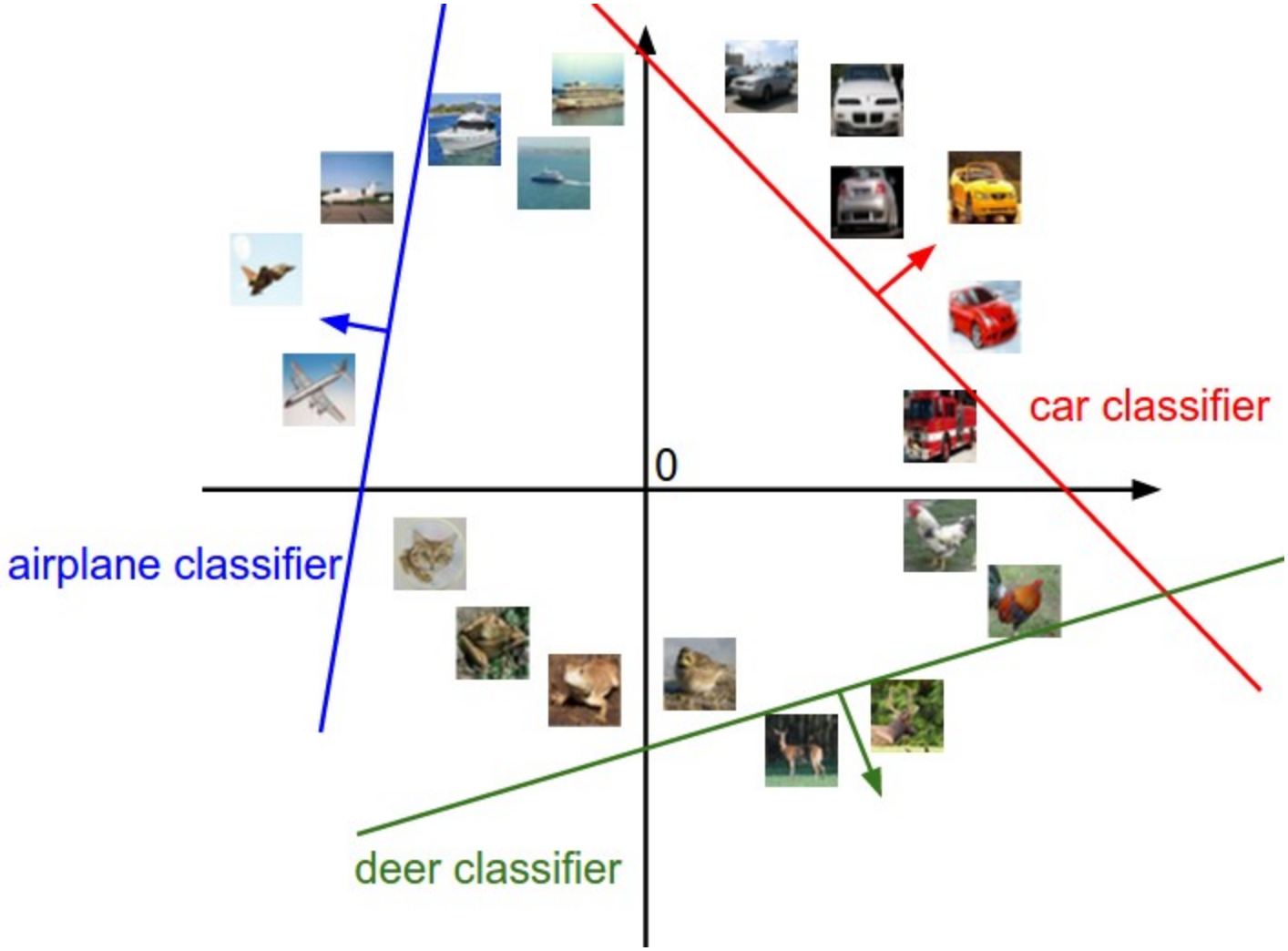
Notice that a linear classifier computes the score of a class as a weighted sum of all of its pixel values across all 3 of its color channels. Depending on precisely what values we set for these weights, the function has the capacity to like or dislike (depending on the sign of each weight) certain colors at certain positions in the image. For instance, you can imagine that the "ship" class might be more likely if there is a lot of blue on the sides of an image (which could likely correspond to water). You might expect that the "ship" classifier would then have a lot of positive weights across its blue channel weights (presence of blue increases score of ship), and negative weights in the red/green channels (presence of red/green decreases the score of ship).



An example of mapping an image to class scores. For the sake of visualization, we assume the image only has 4 pixels (4 monochrome pixels, we are not considering color channels in this example for brevity), 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.

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:



Cartoon representation of the image space, where each image is a single point, and three classifiers are visualized. Using the example of the car classifier (in red), the red line shows all points in the space that get a score of zero for the car class. The red arrow shows the direction of increase, so all points to the right of the red line have positive (and linearly increasing) scores, and all points to the left have a negative (and linearly decreasing) scores.

As we saw above, every row of W is a classifier for one of the classes. The geometric interpretation of these numbers is that as we change one of the rows of W, the corresponding line in the pixel space will rotate in different directions. The biases b, on the other hand, allow our classifiers to translate the lines. In particular, note that without the bias terms, plugging in $x_i = 0$ would always give score of zero regardless of the weights, so all lines would be forced to cross the origin.

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.

Skipping ahead a bit: Example learned weights at the end of learning for CIFAR-10. Note that, for example, the ship template contains a lot of blue pixels as expected. This template will therefore give a high score once it is matched against images of ships on the ocean with an inner product.

Additionally, note that the horse template seems to contain a two-headed horse, which is due to both left and right facing horses in the dataset. The linear classifier *merges* these two modes of horses in the data into a single template. Similarly, the car classifier seems to have merged several modes into a single template which has to identify cars from all sides, and of all colors. In particular, this template ended up being red, which hints that there are more red cars in the CIFAR-10 dataset than of any other color. The linear classifier is too weak to properly account for different-colored cars, but as we will see later neural networks will allow us to perform this task. Looking ahead a bit, a neural network will be able to develop intermediate neurons in its hidden layers that could detect specific car types (e.g. green car facing left, blue car facing front, etc.), and neurons on the next layer could combine these into a more accurate car score through a weighted sum of the individual car detectors.

Bias trick. Before moving on we want to mention a common simplifying trick to representing the two parameters W, b as one. Recall that we defined the score function as:

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

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 x_i 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:

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

With our CIFAR-10 example, x_i is now [3073 x 1] instead of [3072 x 1] - (with the extra dimension holding the constant 1), and W is now [10 x 3073] instead of [10 x 3072]. The extra column that W now corresponds to the bias b. An illustration might help clarify:

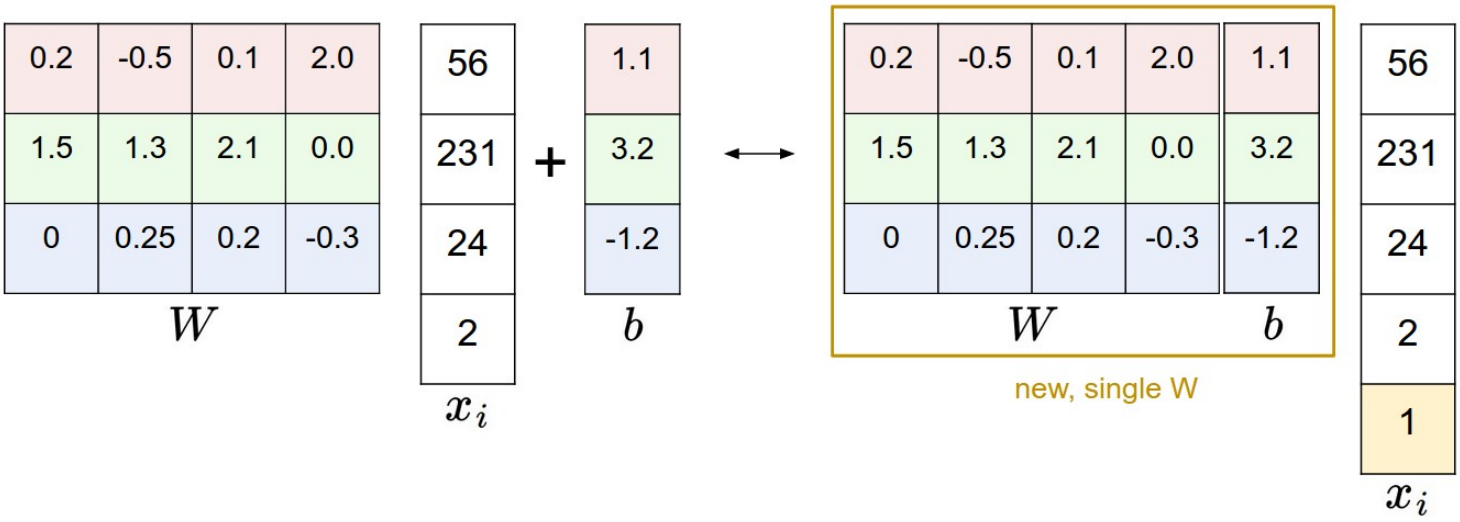


Illustration of the bias trick. Doing a matrix multiplication and then adding a bias vector (left) is equivalent to adding a bias dimension with a constant of 1 to all input vectors and extending the weight matrix by 1 column - a bias column (right). Thus, if we preprocess our data by appending ones to all vectors we only have to learn a single matrix of weights instead of two matrices that hold the weights and the biases.

Image data preprocessing. As a quick note, in the examples above we used the raw pixel values (which range from [0…255]). In Machine Learning, it is a very common practice to always perform normalization of your input features (in the case of images, every pixel is thought of as a feature). 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]. Of these, zero mean centering is arguably more important but we will have to wait for its justification until we understand the dynamics of gradient descent.

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 don't 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.

For example, going back to the example image of a cat and its scores for the classes "cat", "dog" and "ship", we saw that the particular set of weights in that example was not very good at all: We fed in the pixels that depict a cat but the cat score came out very low (-96.8) compared to the other classes (dog score 437.9 and ship score 61.95). We are going to measure our unhappiness with outcomes such as this one with a **loss function** (or sometimes also referred to as the **cost function** or the **objective**). 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.

Multiclass Support Vector Machine loss

There are several ways to define the details of the loss function. As a first example we will first develop a commonly used loss called the **Multiclass Support Vector Machine** (SVM) 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 Δ. Notice that it's sometimes helpful to anthropomorphise the loss functions as we did above: The SVM "wants" a certain outcome in the sense that the outcome would yield a lower loss (which is good).

Let's now get more precise. Recall that for the i-th example 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 (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:

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

Example. Lets unpack this with an example to see how it works. Suppose that we have three classes that receive the scores $s = [13, -7, 11]$, and that the first class is the true class (i.e. $y_i = 0$). Also assume that Δ (a hyperparameter we will go into more detail about soon) is 10. The expression above sums over all incorrect classes ($j \neq y_i$), so we get two terms:

$$L_i = \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10)$$

You can see that the first term gives zero since [-7 - 13 + 10] gives a negative number, which is then thresholded to zero with the $max(0, -)$ function. We get zero loss for this pair because the correct class score (13) was greater than the incorrect class score (-7) by at least the margin 10. In fact the difference was 20, which is much greater than 10 but the SVM only cares that the difference is at least 10; Any additional difference above the margin is clamped at zero with the max operation. The second term computes [11 - 13 + 10] which gives 8. That is, even though the correct class had a higher score than the incorrect class (13 > 11), it was not greater by the desired margin of 10. The difference was only 2, which is why the loss comes out to 8 (i.e. how much higher the difference would have to be to meet the margin). In summary, the SVM loss function wants the score of the correct class y_i to be larger than the incorrect class scores by at least by Δ (delta). If this is not the case, we will accumulate loss.

Note that in this particular module we are working with linear score functions ($f(x_i; W) = Wx_i$), so we can also rewrite the loss function in this equivalent form:

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

where w_j is the j-th row of W reshaped as a column. However, this will not necessarily be the case once we start to consider more complex forms of the score function f.

A last piece of terminology we'll mention before we finish with this section is that the threshold at zero $max(0, -)$ function is often called the **hinge loss**. You'll sometimes hear about people instead using the squared hinge loss SVM (or L2-SVM), which uses the form $max(0, -)^2$ that penalizes violated margins more strongly (quadratically instead of linearly). The unsquared version is more standard, but in some datasets the squared hinge loss can work better. This can be determined during cross-validation.

> *The loss function quantifies our unhappiness with predictions on the training set*

The Multiclass Support Vector Machine "wants" the score of the correct class to be higher than all other scores by at least a margin of delta. If any class has a score inside the red region (or higher), then there will be accumulated loss. Otherwise the loss will be zero. Our objective will be to find the weights that will simultaneously satisfy this constraint for all examples in the training data and give a total loss that is as low as possible.

Regularization. There is one bug with the loss function we presented above. 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 λW where $\lambda > 1$ will also give zero loss because this transformation uniformly stretches all score magnitudes and hence also their absolute differences. For example, if the difference in scores between a correct class and a nearest incorrect class was 15, then multiplying all elements of **W** by 2 would make the new difference 30.

In other words, 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 squared **L2** norm that discourages large weights through an elementwise quadratic penalty over all parameters:

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

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 L_i over all examples) and the **regularization loss**. That is, the full Multiclass SVM loss becomes:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

Or expanding this out in its full form:

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda \sum_k \sum_l W_{k,l}^2$$

Where N is the number of training examples. As you can see, we append the regularization penalty to the loss objective, weighted by a hyperparameter λ. There is no simple way of setting this hyperparameter and it is usually determined by cross-validation.

In addition to the motivation we provided above there are many desirable properties to include the regularization penalty, many of which we will come back to in later sections. For example, it turns out that including the L2 penalty leads to the appealing **max margin** property in SVMs (See CS229 lecture notes for full details if you are interested).

The most appealing property is that penalizing large weights tends to improve generalization, because it means that no input dimension can have a very large influence on the scores all by itself. For example, suppose that we have some input vector $x = [1, 1, 1, 1]$ and two weight vectors $w_1 = [1, 0, 0, 0]$, $w_2 = [0.25, 0.25, 0.25, 0.25]$. Then $w_1^T x = w_2^T x = 1$ so both weight vectors lead to the same dot product, but the L2 penalty of w_1 is 1.0 while the L2 penalty of w_2 is only 0.5. Therefore, according to the L2 penalty the weight vector w_2 would be preferred since it achieves a lower regularization loss. Intuitively, this is because the weights in w_2 are smaller and more diffuse. Since the L2 penalty prefers smaller and more diffuse weight vectors, the final classifier is encouraged to take into account all input dimensions to small amounts rather than a few input dimensions and very strongly. As we will see later in the class, this effect can improve the generalization performance of the classifiers on test images and lead to less *overfitting*.

Note that biases do not have the same effect since, unlike the weights, they do not control the strength of influence of an input dimension. Therefore, it is common to only regularize the weights W but not the biases b. However, in practice this often turns out to have a negligible effect. Lastly, note that due to the regularization penalty we can never achieve loss of exactly 0.0 on all examples, because this would only be possible in the pathological setting of $W = 0$.

Code. Here is the loss function (without regularization) implemented in Python, in both unvectorized and half-vectorized form:

```
def L_i(x, y, W):
  """
  unvectorized version. Compute the multiclass svm loss for a single example (x,y)
  - x is a column vector representing an image (e.g. 3073 x 1 in CIFAR-10)
    with an appended bias dimension in the 3073-rd position (i.e. bias trick)
```

```
  - y is an integer giving index of correct class (e.g. between 0 and 9 in CIFAR-10
  - W is the weight matrix (e.g. 10 x 3073 in CIFAR-10)
  """
  delta = 1.0 # see notes about delta later in this section
  scores = W.dot(x) # scores becomes of size 10 x 1, the scores for each class
  correct_class_score = scores[y]
  D = W.shape[0] # number of classes, e.g. 10
  loss_i = 0.0
  for j in range(D): # iterate over all wrong classes
    if j == y:
      # skip for the true class to only loop over incorrect classes
      continue
    # accumulate loss for the i-th example
    loss_i += max(0, scores[j] - correct_class_score + delta)
  return loss_i

def L_i_vectorized(x, y, W):
  """
  A faster half-vectorized implementation. half-vectorized
  refers to the fact that for a single example the implementation contains
  no for loops, but there is still one loop over the examples (outside this functio
  """
  delta = 1.0
  scores = W.dot(x)
  # compute the margins for all classes in one vector operation
  margins = np.maximum(0, scores - scores[y] + delta)
  # on y-th position scores[y] - scores[y] canceled and gave delta. We want
  # to ignore the y-th position and only consider margin on max wrong class
  margins[y] = 0
  loss_i = np.sum(margins)
  return loss_i

def L(X, y, W):
  """
  fully-vectorized implementation :
  - X holds all the training examples as columns (e.g. 3073 x 50,000 in CIFAR-10)
  - y is array of integers specifying correct class (e.g. 50,000-D array)
  - W are weights (e.g. 10 x 3073)
  """
  # evaluate loss over all examples in X without using any for loops
  # left as exercise to reader in the assignment
```

The takeaway from this section is that the SVM loss takes one particular approach to measuring how consistent the predictions on training data are with the ground truth labels. Additionally, making good predictions on the training set is equivalent to minimizing the loss.

All we have to do now is to come up with a way to find the weights that minimize the loss.

Practical Considerations

Setting Delta. Note that we brushed over the hyperparameter Δ and its setting. What value should it be set to, and do we have to cross-validate it? It turns out that this hyperparameter can safely be set to $\Delta = 1.0$ in all cases. The hyperparameters Δ and λ seem like two different hyperparameters, but in fact they both control the same tradeoff: The tradeoff between the data loss and the regularization loss in the objective. The key to understanding this is that the magnitude of the weights W has direct effect on the scores (and hence also their differences): As we shrink all values inside W the score differences will become lower, and as we scale up the weights the score differences will all become higher. Therefore, the exact value of the margin between the scores (e.g. $\Delta = 1$, or $\Delta = 100$) is in some sense meaningless because the weights can shrink or stretch the differences arbitrarily. Hence, the only real tradeoff is how large we allow the weights to grow (through the regularization strength λ).

Relation to Binary Support Vector Machine. You may be coming to this class with previous experience with Binary Support Vector Machines, where the loss for the i-th example can be written as:

$$L_i = C \max(0, 1 - y_i w^T x_i) + R(W)$$

where C is a hyperparameter, and $y_i \in \{-1, 1\}$. You can convince yourself that the formulation we presented in this section contains the binary SVM as a special case when there are only two classes. That is, if we only had two classes then the loss reduces to the binary SVM shown above. Also, C in this formulation and λ in our formulation control the same tradeoff and are related through reciprocal relation $C \propto \frac{1}{\lambda}$.

Aside: Optimization in primal. If you're coming to this class with previous knowledge of SVMs, you may have also heard of kernels, duals, the SMO algorithm, etc. In this class (as is the case with Neural Networks in general) we will always work with the optimization objectives in their unconstrained primal form. Many of these objectives are technically not differentiable (e.g. the max(x,y) function isn't because it has a *kink* when x=y), but in practice this is not a problem and it is common to use a subgradient.

Aside: Other Multiclass SVM formulations. It is worth noting that the Multiclass SVM presented in this section is one of few ways of formulating the SVM over multiple classes. Another commonly used form is the *One-Vs-All* (OVA) SVM which trains an independent binary SVM for each class vs. all other classes. Related, but less common to see in practice is also the *All-vs-All* (AVA) strategy. Our formulation follows the Weston and Watkins 1999 (pdf) version, which is a more powerful version than OVA (in the sense that you can construct multiclass datasets where this version can achieve zero data loss, but OVA cannot. See details in the paper if interested). The last formulation you may see is a *Structured SVM*, which maximizes the margin between the score of the correct class and the score of the highest-scoring incorrect runner-up class. Understanding the differences between these formulations is outside of the scope of the class. The version presented in these notes is a safe bet to use in practice, but the arguably simplest OVA strategy is likely to work just as well (as also argued by Rikin et al. 2004 in In Defense of One-Vs-All Classification (pdf)).

Softmax classifier

It turns out that the SVM is one of two commonly seen classifiers. The other popular choice is the **Softmax classifier**, which has a different loss function. If you've heard of the binary Logistic Regression classifier before, the Softmax classifier is its generalization to multiple classes. 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:

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

where we are using the notation f_j to mean the j-th element of the vector of class scores f. As before, the full loss for the dataset is the mean of L_i over all training examples together with a regularization term $R(W)$. The function $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$ is called 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. The full cross-entropy loss that involves the softmax function might look scary if you're seeing it for the first time but it is relatively easy to motivate.

Information theory view. The *cross-entropy* between a "true" distribution p and an estimated distribution q is defined as:

$$H(p, q) = -\sum_x p(x)\log q(x)$$

The Softmax classifier is hence minimizing the cross-entropy between the estimated class probabilities ($q = e^{f_{y_i}} / \sum_j e^{f_j}$ as seen above) and the "true" distribution, which in this interpretation is the distribution where all probability mass is on the correct class (i.e. $p = [0, \ldots 1, \ldots, 0]$ contains a single 1 at the y_i -th position.). Moreover, since the cross-entropy can be written in terms of entropy and the Kullback-Leibler divergence as $H(p, q) = H(p) + D_{KL}(p||q)$, and the entropy of the delta function p is zero, this is also equivalent to minimizing the KL divergence between the two distributions (a measure of distance). In other words, the cross-entropy objective *wants* the predicted distribution to have all of its mass on the correct answer.

Probabilistic interpretation. Looking at the expression, we see that

$$P(y_i \mid x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

can be interpreted as the (normalized) probability assigned to the correct label y_i given the image x_i and

emember that the Softmax classifier interprets the scores inside the

output vector f as the unnormalized log probabilities. Exponentiating these quantities therefore gives the (unnormalized) probabilities, and the division performs the normalization so that the probabilities sum to one. In the probabilistic interpretation, we are therefore minimizing the negative log likelihood of the correct class, which can be interpreted as performing *Maximum Likelihood Estimation* (MLE). A nice feature of this view is that we can now also interpret the regularization term $R(W)$ in the full loss function as coming from a Gaussian prior over the weight matrix W, where instead of MLE we are performing the *Maximum a posteriori* (MAP) estimation. We mention these interpretations to help your intuitions, but the full details of this derivation are beyond the scope of this class.

Practical issues: Numeric stability. When you're writing code for computing the Softmax function in practice, the intermediate terms $e^{f_{y_i}}$ and $\sum_j e^{f_j}$ may be very large due to the exponentials. Dividing large numbers can be numerically unstable, so it is important to use a normalization trick. Notice that if we multiply the top and bottom of the fraction by a constant C and push it into the sum, we get the following (mathematically equivalent) expression:

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i} + \log C}}{\sum_j e^{f_j + \log C}}$$

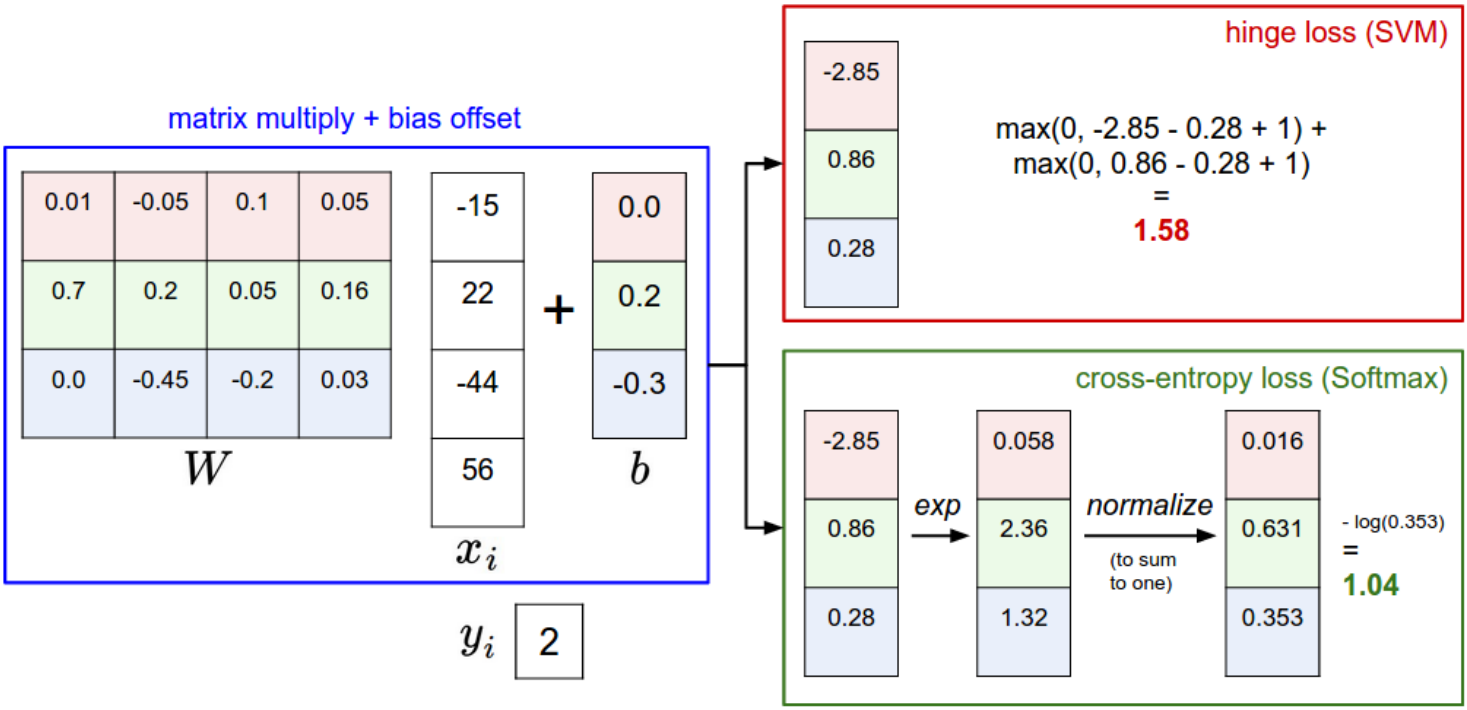
We are free to choose the value of C. This will not change any of the results, but we can use this value to improve the numerical stability of the computation. A common choice for C is to set $\log C = -\max_j f_j$. This simply states that we should shift the values inside the vector f so that the highest value is zero. In code:

```
f = np.array([123, 456, 789]) # example with 3 classes and each having large scores
p = np.exp(f) / np.sum(np.exp(f)) # Bad: Numeric problem, potential blowup

# instead: first shift the values of f so that the highest number is 0:
f -= np.max(f) # f becomes [-666, -333, 0]
p = np.exp(f) / np.sum(np.exp(f)) # safe to do, gives the correct answer
```

Possibly confusing naming conventions. To be precise, the *SVM classifier* uses the *hinge loss*, or also sometimes called the *max-margin loss*. The *Softmax classifier* uses the *cross-entropy loss*. The Softmax classifier gets its name from the *softmax function*, which is used to squash the raw class scores into normalized positive values that sum to one, so that the cross-entropy loss can be applied. In particular, note that technically it doesn't make sense to talk about the "softmax loss", since softmax is just the squashing function, but it is a relatively commonly used shorthand.

SVM vs. Softmax

A picture might help clarify the distinction between the Softmax and SVM classifiers:



Example of the difference between the SVM and Softmax classifiers for one datapoint. In both cases we compute the same score vector **f** (e.g. by matrix multiplication in this section). The difference is in the interpretation of the scores in **f**: The SVM interprets these as class scores and its loss function encourages the correct class (class 2, in blue) to have a score higher by a margin than the other class scores. The Softmax classifier instead interprets the scores as (unnormalized) log probabilities for each class and then encourages the (normalized) log probability of the correct class to be high (equivalently the negative of it to be low). The final loss for this example is 1.58 for the SVM and 1.04 (note this is 1.04 using the natural logarithm, not base 2 or base 10) for the Softmax classifier, but note that these numbers are not comparable; They are only meaningful in relation to loss computed within the same classifier and with the same data.

Softmax classifier provides "probabilities" for each class. 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. For example, given an image the SVM classifier might give you scores [12.5, 0.6, -23.0] for the classes "cat", "dog" and "ship". The softmax classifier can instead compute the probabilities of the three labels as [0.9, 0.09, 0.01], which allows you to interpret its confidence in each class. The reason we put the word "probabilities" in quotes, however, is that how peaky or diffuse these probabilities are depends directly on the regularization strength λ - which you are in charge of as input to the system. For example, suppose that the unnormalized log-probabilities for some three classes come out to be [1, -2, 0]. The softmax function would then compute:

$$[1, -2, 0] \rightarrow [e^1, e^{-2}, e^0] = [2.71, 0.14, 1] \rightarrow [0.7, 0.04, 0.26]$$

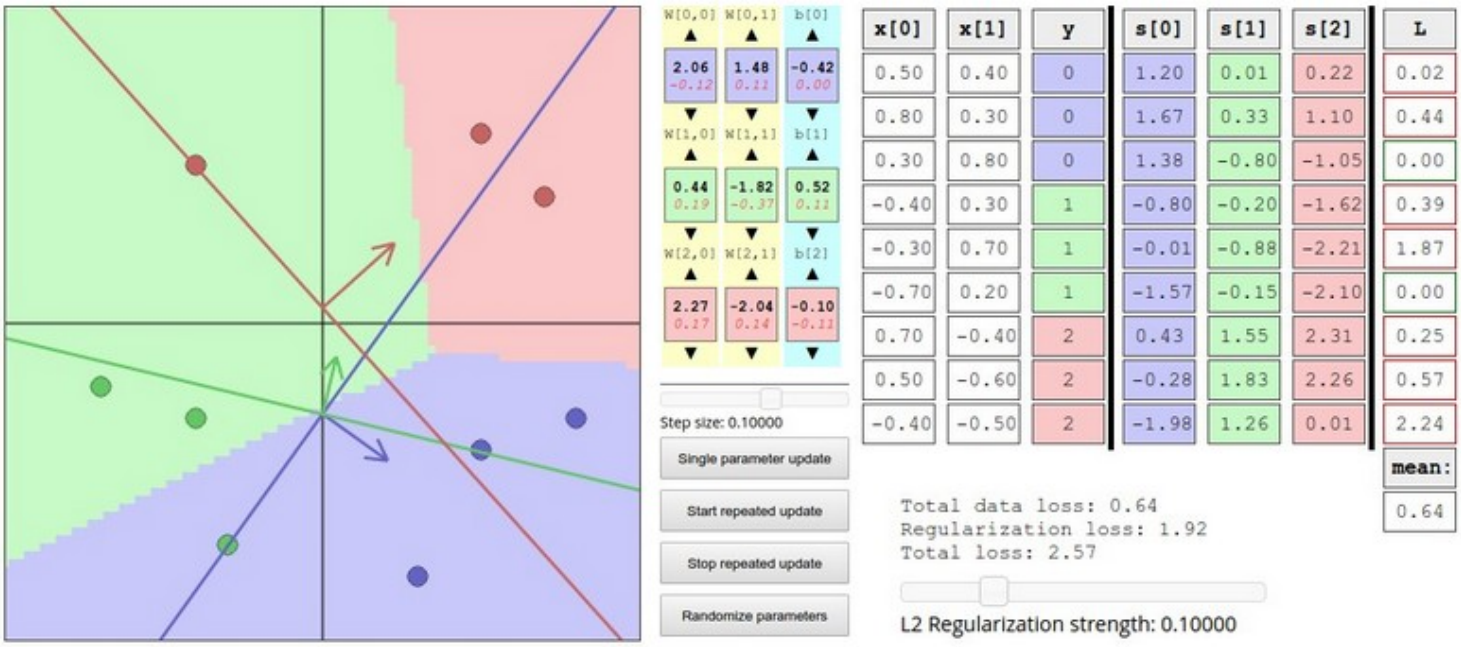
Where the steps taken are to exponentiate and normalize to sum to one. Now, if the regularization strength λ was higher, the weights W would be penalized more and this would lead to smaller weights. For example, suppose that the weights became one half smaller ([0.5, -1, 0]). The softmax would now compute:

$$[0.5, -1, 0] \rightarrow [e^{0.5}, e^{-1}, e^0] = [1.65, 0.37, 1] \rightarrow [0.55, 0.12, 0.33]$$

where the probabilites are now more diffuse. Moreover, in the limit where the weights go towards tiny numbers due to very strong regularization strength λ, the output probabilities would be near uniform. Hence, the probabilities computed by the Softmax classifier are better thought of as confidences where, similar to the SVM, the ordering of the scores is interpretable, but the absolute numbers (or their differences) technically are not.

In practice, SVM and Softmax are usually comparable. The performance difference between the SVM and Softmax are usually very small, and different people will have different opinions on which classifier works better. Compared to the Softmax classifier, the SVM is a more *local* objective, which could be thought of either as a bug or a feature. Consider an example that achieves the scores [10, -2, 3] and where the first class is correct. An SVM (e.g. with desired margin of $\Delta = 1$) will see that the correct class already has a score higher than the margin compared to the other classes and it will compute loss of zero. The SVM does not care about the details of the individual scores: if they were instead [10, -100, -100] or [10, 9, 9] the SVM would be indifferent since the margin of 1 is satisfied and hence the loss is zero. However, these scenarios are not equivalent to a Softmax classifier, which would accumulate a much higher loss for the scores [10, 9, 9] than for [10, -100, -100]. 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. This can intuitively be thought of as a feature: For example, a car classifier which is likely spending most of its "effort" on the difficult problem of separating cars from trucks should not be influenced by the frog examples, which it already assigns very low scores to, and which likely cluster around a completely different side of the data cloud.

Interactive web demo



We have written an interactive web demo to help your intuitions with linear classifiers. The demo visualizes the loss functions discussed in this section using a toy 3-way classification on 2D data. The demo also jumps ahead a bit and performs the optimization, which we will discuss in full detail in the next section.

Summary

In summary,

- We defined a **score function** from image pixels to class scores (in this section, a linear function that depends on weights **W** and biases **b**).
- Unlike kNN classifier, the advantage of this **parametric approach** is that once we learn the parameters we can discard the training data. Additionally, the prediction for a new test image is fast since it requires a single matrix multiplication with **W**, not an exhaustive comparison to every single training example.
- We introduced the **bias trick**, which allows us to fold the bias vector into the weight matrix for convenience of only having to keep track of one parameter matrix.
- We defined a **loss function** (we introduced two commonly used losses for linear classifiers: the **SVM** and the **Softmax**) that measures how compatible a given set of parameters is with respect to the ground truth labels in the training dataset. We also saw that the loss function was defined in such way that making good predictions on the training data is equivalent to having a small loss.

We now saw one way to take a dataset of images and map each one to class scores based on a set of parameters, and we saw two examples of loss functions that we can use to measure the quality of the predictions. But how do we efficiently determine the parameters that give the best (lowest) loss? This process is *optimization*, and it is the topic of the next section.

Further Reading

These readings are optional and contain pointers of interest.

- Deep Learning using Linear Support Vector Machines from Charlie Tang 2013 presents some results claiming that the L2SVM outperforms Softmax.

Table of Contents:

Introduction

In the previous section we introduced two key components in context of the image classification task:

1. A (parameterized) **score function** mapping the raw image pixels to class scores (e.g. a linear function)
2. A **loss function** that measured the quality of a particular set of parameters based on how well the induced scores agreed with the ground truth labels in the training data. We saw that there are many ways and versions of this (e.g. Softmax/SVM).

Concretely, recall that the linear function had the form $f(x_i, W) = Wx_i$ and the SVM we developed was formulated as:

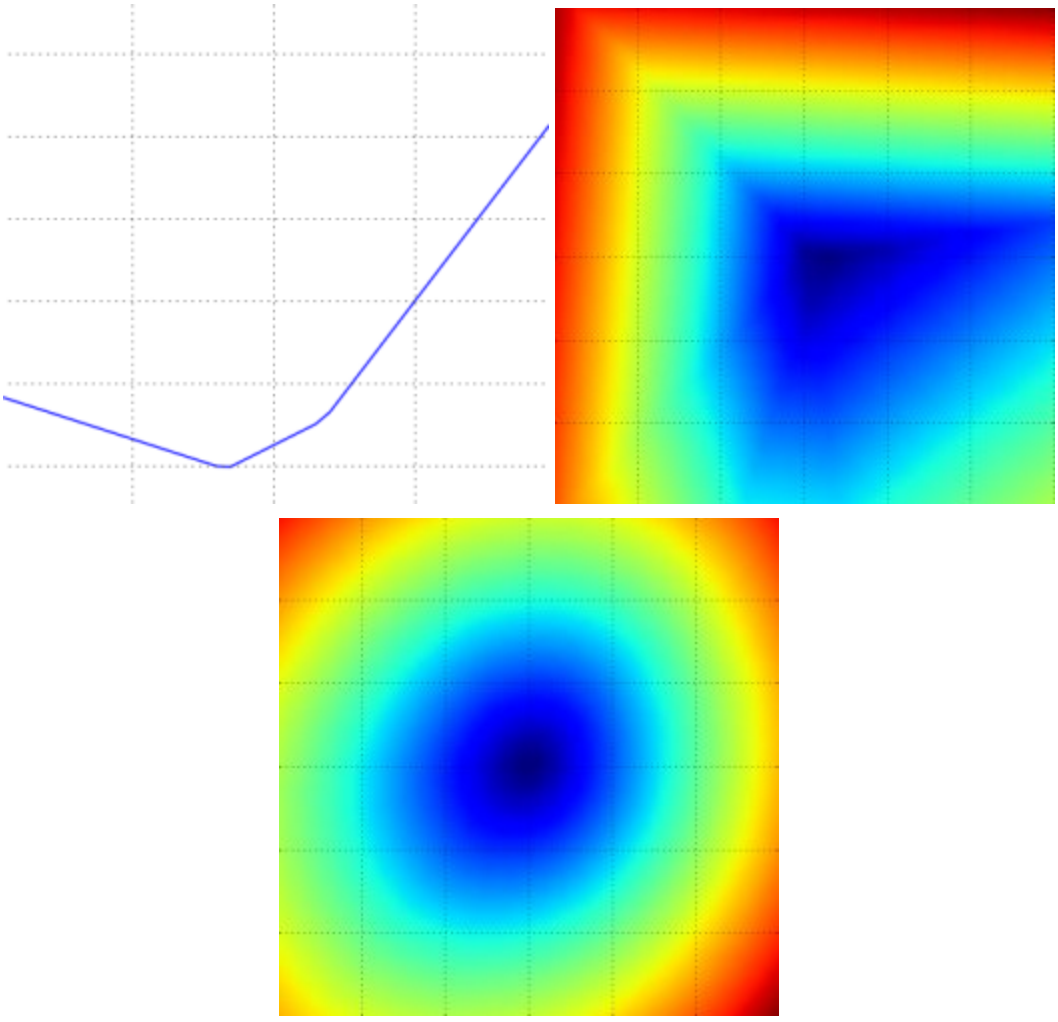
$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max\left(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1\right) \right] + \alpha R(W)$$

We saw that a setting of the parameters W that produced predictions for examples x_i consistent with their ground truth labels y_i would also have a very low loss L. 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.

Foreshadowing: Once we understand how these three core components interact, we will revisit the first component (the parameterized function mapping) and extend it to functions much more complicated than a linear mapping: First entire Neural Networks, and then Convolutional Neural Networks. The loss functions and the optimization process will remain relatively unchanged.

Visualizing the loss function

The loss functions we'll 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). For example, we can generate a random weight matrix W (which corresponds to a single point in the space), then march along a ray and record the loss function value along the way. That is, we can generate a random direction W_1 and compute the loss along this direction by evaluating $L(W + aW_1)$ for different values of a. This process generates a simple plot with the value of a as the x-axis and the value of the loss function as the y-axis. We can also carry out the same procedure with two dimensions by evaluating the loss $L(W + aW_1 + bW_2)$ as we vary a, b. In a plot, a, b could then correspond to the x-axis and the y-axis, and the value of the loss function can be visualized with a color:



Loss function landscape for the Multiclass SVM (without regularization) for one single example (left,middle) and for a (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).

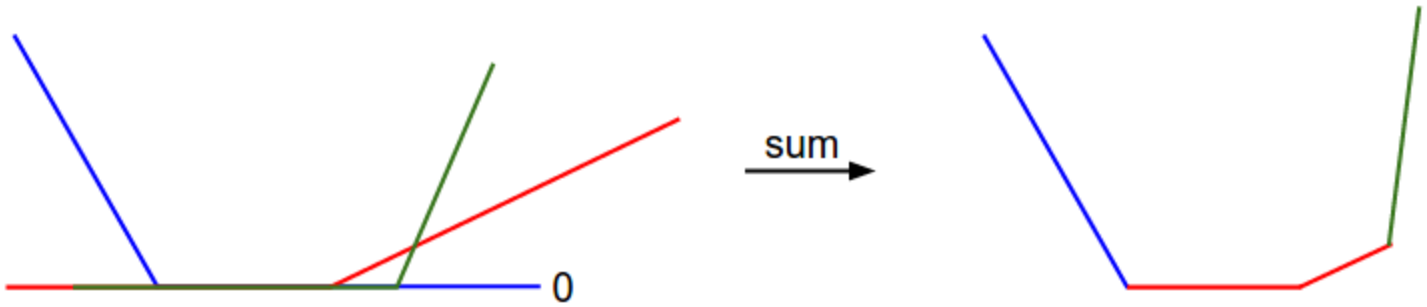
We can explain the piecewise-linear structure of the loss function by examining the math. For a single example we have:

$$L_i = \sum_{j \neq y_i} \left[\max\left(0, w_j^T x_i - w_{y_i}^T x_i + 1\right) \right]$$

It is clear from the equation that the data loss for each example is a sum of (zero-thresholded due to the $\max(0, -)$ function) linear functions of W. Moreover, each row of W (i.e. w_j) sometimes has a positive sign in front of it (when it corresponds to a wrong class for an example), and sometimes a negative sign (when it corresponds to the correct class for that example). To make this more explicit, consider a simple dataset that contains three 1-dimensional points and three classes. The full SVM loss (without regularization) becomes:

$$\begin{aligned}
L_0 &= \max\left(0, w_1^T x_0 - w_0^T x_0 + 1\right) + \max\left(0, w_2^T x_0 - w_0^T x_0 + 1\right) \\
L_1 &= \max\left(0, w_0^T x_1 - w_1^T x_1 + 1\right) + \max\left(0, w_2^T x_1 - w_1^T x_1 + 1\right) \\
L_2 &= \max\left(0, w_0^T x_2 - w_2^T x_2 + 1\right) + \max\left(0, w_1^T x_2 - w_2^T x_2 + 1\right) \\
L &= (L_0 + L_1 + L_2)/3
\end{aligned}$$

Since these examples are 1-dimensional, the data x_i and weights w_j are numbers. Looking at, for instance, w_0, some terms above are linear functions of w_0 and each is clamped at zero. We can visualize this as follows:



1-dimensional illustration of the data loss. The x-axis is a single weight and the y-axis is the loss. The data loss is a sum of multiple terms, each of which is either independent of a particular weight, or a linear function of it that is thresholded at zero. The full SVM data loss is a 30,730-dimensional version of this shape.

As an aside, you may have guessed from its bowl-shaped appearance that the SVM cost function is an example of a convex function There is a large amount of literature devoted to efficiently minimizing these types of functions, and you can also take a Stanford class on the topic (convex optimization). Once we

extend our score functions f to Neural Networks our objective functions will become non-convex, and the visualizations above will not feature bowls but complex, bumpy terrains.

Non-differentiable loss functions. As a technical note, you can also see that the *kinks* in the loss function (due to the max operation) technically make the loss function non-differentiable because at these kinks the gradient is not defined. However, the subgradient still exists and is commonly used instead. In this class will use the terms *subgradient* and *gradient* interchangeably.

Optimization

To reiterate, the loss function lets us quantify the quality of any particular set of weights **W**. The goal of optimization is to find **W** that minimizes the loss function. We will now motivate and slowly develop an approach to optimizing the loss function. For those of you coming to this class with previous experience, this section might seem odd since the working example we'll use (the SVM loss) is a convex problem, but keep in mind that our goal is to eventually optimize Neural Networks where we can't easily use any of the tools developed in the Convex Optimization literature.

Strategy #1: A first very bad idea solution: 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. This procedure might look as follows:

```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in range(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
```

```
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```

In the code above, we see that we tried out several random weight vectors **W**, and some of them work better than others. We can take the best weights **W** found by this search and try it out on the test set:

```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

With the best **W** this gives an accuracy of about **15.5%**. Given that guessing classes completely at random achieves only 10%, that's not a very bad outcome for a such a brain-dead random search solution!

Core idea: iterative refinement. Of course, it turns out that we can do much better. The core idea is that finding the best set of weights **W** is a very difficult or even impossible problem (especially once **W** contains weights for entire complex neural networks), but the problem of refining a specific set of weights **W** to be slightly better is significantly less difficult. In other words, our approach will be to start with a random **W** and then iteratively refine it, making it slightly better each time.

> *Our strategy will be to start with random weights and iteratively refine them over time to get lower loss*

Blindfolded hiker analogy. One analogy that you may find helpful going forward is to think of yourself as hiking on a hilly terrain with a blindfold on, and trying to reach the bottom. In the example of CIFAR-10, the hills are 30,730-dimensional, since the dimensions of **W** are 10 x 3073. At every point on the hill we achieve a particular loss (the height of the terrain).

Strategy #2: 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 δW to it and if the loss at the perturbed $W + \delta W$ is lower, we will perform an update. The code for this procedure is as follows:

```
W = np.random.randn(10, 3073) * 0.001 # generate random starting W
bestloss = float("inf")
for i in range(1000):
  step_size = 0.0001
  Wtry = W + np.random.randn(10, 3073) * step_size
  tr_cols, Ytr, Wtry)
```

```
  if loss < bestloss:
    W = Wtry
    bestloss = loss
  print 'iter %d loss is %f' % (i, bestloss)
```

Using the same number of loss function evaluations as before (1000), this approach achieves test set classification accuracy of **21.4%**. This is better, but still wasteful and computationally expensive.

Strategy #3: Following the Gradient

In the previous section we tried to find a direction in the weight-space that would improve our weight vector (and give us a lower loss). It turns out that there is no need to randomly search for a good direction: 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 descent (at least in the limit as the step size goes towards zero). This direction will be related to the **gradient** of the loss function. In our hiking analogy, this approach roughly corresponds to feeling the slope of the hill below our feet and stepping down the direction that feels steepest.

In one-dimensional functions, the slope is the instantaneous rate of change of the function at any point you might be interested in. The gradient is a generalization of slope for functions that don't take a single number but a vector of numbers. Additionally, the gradient is just a vector of slopes (more commonly referred to as **derivatives**) for each dimension in the input space. The mathematical expression for the derivative of a 1-D function with respect its input is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

When the functions of interest take a vector of numbers instead of a single number, we call the derivatives **partial derivatives**, and the gradient is simply the vector of partial derivatives in each dimension.

Computing the gradient

There are two ways to compute the gradient: A slow, approximate but easy way (**numerical gradient**), and a fast, exact but more error-prone way that requires calculus (**analytic gradient**). We will now present both.

Computing the gradient numerically with finite differences

The formula given above allows us to compute the gradient numerically. Here is a generic function that takes a function `f`, a vector `x` to evaluate the gradient on, and returns the gradient of `f` at `x`:

```
def eval_numerical_gradient(f, x):
  """
  plementation of numerical gradient of f at x
```

```
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

Following the gradient formula we gave above, 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. The variable `grad` holds the full gradient in the end.

Practical considerations. Note that in the mathematical formulation the gradient is defined in the limit as **h** goes towards zero, but in practice it is often sufficient to use a very small value (such as 1e-5 as seen in the example). Ideally, you want to use the smallest step size that does not lead to numerical issues. Additionally, in practice it often works better to compute the numeric gradient using the **centered difference formula**: $[f(x+h) - f(x-h)]/2h$. See wiki for details.

We can use the function given above to compute the gradient at any point and for any function. Lets compute the gradient for the CIFAR-10 loss function at some random point in the weight space:

```
# to use the generic code above we want a function that takes a single argument
# (the weights in our case) so we close over X_train and Y_train
def CIFAR10_loss_fun(W):
  return L(X_train, Y_train, W)
```

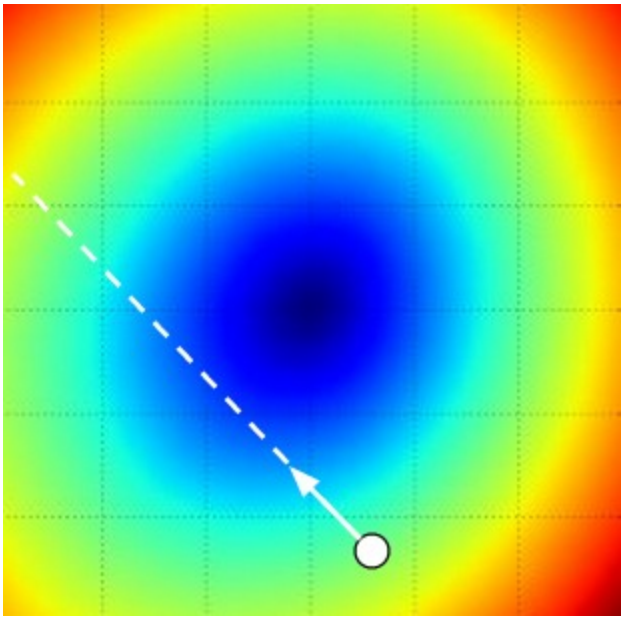
```
W = np.random.rand(10, 3073) * 0.001 # random weight vector
df = eval_numerical_gradient(CIFAR10_loss_fun, W) # get the gradient
```

The gradient tells us the slope of the loss function along every dimension, which we can use to make an update:

```
loss_original = CIFAR10_loss_fun(W) # the original loss
print 'original loss: %f' % (loss_original, )

# lets see the effect of multiple step sizes
for step_size_log in [-10, -9, -8, -7, -6, -5,-4,-3,-2,-1]:
  step_size = 10 ** step_size_log
  W_new = W - step_size * df # new position in the weight space
  loss_new = CIFAR10_loss_fun(W_new)
  print 'for step size %f new loss: %f' % (step_size, loss_new)

# prints:
# original loss: 2.200718
# for step size 1.000000e-10 new loss: 2.200652
# for step size 1.000000e-09 new loss: 2.200057
# for step size 1.000000e-08 new loss: 2.194116
# for step size 1.000000e-07 new loss: 2.135493
# for step size 1.000000e-06 new loss: 1.647802
# for step size 1.000000e-05 new loss: 2.844355
# for step size 1.000000e-04 new loss: 25.558142
# for step size 1.000000e-03 new loss: 254.086573
# for step size 1.000000e-02 new loss: 2539.370888
# for step size 1.000000e-01 new loss: 25392.214036
```

Update in negative gradient direction. In the code above, notice that to compute `W_new` we are making an update in the negative direction of the gradient `df` since we wish our loss function to decrease, not increase.

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. In our blindfolded hill-descent analogy, we feel the hill below our feet sloping in some direction, but the step length we should take is uncertain. If we shuffle our feet carefully we can expect to make consistent but very small progress (this corresponds to having a small step size). Conversely, we can choose to make a large, confident step in an attempt to descend faster, but this may not pay off. As you can see in the code example above, at some point taking a bigger step gives a higher loss as we "overstep".



Visualizing the effect of step size. We start at some particular spot W and evaluate the gradient (or rather its negative - the white arrow) which tells us the direction of the steepest decrease in the loss function. Small steps are likely to lead to consistent but slow progress. Large steps can lead to better progress but are more risky. Note that eventually, for a large step size we will overshoot and make the loss worse. The step size (or as we will later call it - the **learning rate**) will become one of the most important hyperparameters that we will have to carefully tune.

A problem of efficiency. You may have noticed that evaluating the numerical gradient has complexity linear in the number of parameters. In our example we had 30730 parameters in total and therefore had to perform 30,731 evaluations of the loss function to evaluate the gradient and to perform only a single parameter update. This problem only gets worse, since modern Neural Networks can easily have tens of millions of parameters. Clearly, this strategy is not scalable and we need something better.

Computing the gradient analytically with Calculus

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**.

Lets use the example of the SVM loss function for a single datapoint:

$$L_i = \sum_{j \neq y_i} \left[\max\left(0, w_j^T x_i - w_{y_i}^T x_i + \Delta\right) \right]$$

We can differentiate the function with respect to the weights. For example, taking the gradient with respect to w_{y_i} we obtain:

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i$$

where 1 is the indicator function that is one if the condition inside is true or zero otherwise. While the expression may look scary when it is written out, when you're implementing this in code you'd simply count the number of classes that didn't meet the desired margin (and hence contributed to the loss function) and then the data vector x_i scaled by this number is the gradient. Notice that this is the gradient only with respect to the row of W that corresponds to the correct class. For the other rows where $j \neq y_i$ the gradient is:

$$\nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

Once you derive the expression for the gradient it is straight-forward to implement the expressions and use them to perform the gradient update.

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:

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

This simple loop is at the core of all Neural Network libraries. There are other ways of performing the optimization (e.g. LBFGS), but Gradient Descent is currently by far the most common and established way of optimizing Neural Network loss functions. Throughout the class we will put some bells and whistles on the details of this loop (e.g. the exact details of the update equation), but the core idea of following the gradient until we're happy with the results will remain the same.

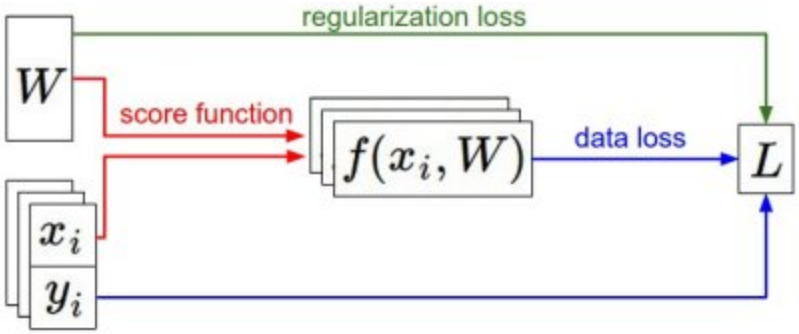
Mini-batch gradient descent. In large-scale applications (such as the ILSVRC challenge), the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over **batches** of the training data. For example, in current state of the art ConvNets, a typical batch contains 256 examples from the entire training set of 1.2 million. This batch is then used to perform a parameter update:

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

The reason this works well is that the examples in the training data are correlated. To see this, consider the extreme case where all 1.2 million images in ILSVRC are in fact made up of exact duplicates of only 1000 unique images (one for each class, or in other words 1200 identical copies of each image). Then it is clear that the gradients we would compute for all 1200 identical copies would all be the same, and when we average the data loss over all 1.2 million images we would get the exact same loss as if we only evaluated on a small subset of 1000. In practice of course, the dataset would not contain duplicate images, the gradient from a mini-batch is a good approximation of the gradient of the full objective. Therefore, much faster convergence can be achieved in practice by evaluating the mini-batch gradients to perform more frequent parameter updates.

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. Even though SGD technically refers to using a single example at a time to evaluate the gradient, you will hear people use the term SGD even when referring to mini-batch gradient descent (i.e. mentions of MGD for "Minibatch Gradient Descent", or BGD for "Batch gradient descent" are rare to see), where it is usually assumed that mini-batches are used. 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.

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.

In this section,

- We developed the intuition of the loss function as a **high-dimensional optimization landscape** in which we are trying to reach the bottom. The working analogy we developed was that of a blindfolded hiker

who wishes to reach the bottom. In particular, we saw that the SVM cost function is piece-wise linear and bowl-shaped.

- We motivated the idea of optimizing the loss function with **iterative refinement**, where we start with a random set of weights and refine them step by step until the loss is minimized.
- We saw that the **gradient** of a function gives the steepest ascent direction and we discussed a simple but inefficient way of computing it numerically using the finite difference approximation (the finite difference being the value of *h* used in computing the numerical gradient).
- We saw that the parameter update requires a tricky setting of the **step size** (or the **learning rate**) that must be set just right: if it is too low the progress is steady but slow. If it is too high the progress can be faster, but more risky. We will explore this tradeoff in much more detail in future sections.
- We discussed the tradeoffs between computing the **numerical** and **analytic** gradient. The numerical gradient is simple but it is approximate and expensive to compute. The analytic gradient is exact, fast to compute but more error-prone since it requires the derivation of the gradient with math. Hence, in practice we always use the analytic gradient and then perform a **gradient check**, in which its implementation is compared to the numerical gradient.
- We introduced the **Gradient Descent** algorithm which iteratively computes the gradient and performs a parameter update in loop.

Coming up: The core takeaway from this section is that the ability to compute the gradient of a loss function with respect to its weights (and have some intuitive understanding of it) is the most important skill needed to design, train and understand neural networks. In the next section we will develop proficiency in computing the gradient analytically using the chain rule, otherwise also referred to as **backpropagation**. This will allow us to efficiently optimize relatively arbitrary loss functions that express all kinds of Neural Networks, including Convolutional Neural Networks.

Table of Contents:

Introduction

Motivation. In this section we will develop expertise with an intuitive understanding of **backpropagation**, which is a way of computing gradients of expressions through recursive application of **chain rule**. Understanding of this process and its subtleties is critical for you to understand, and effectively develop, design and debug neural networks.

Problem statement. The core problem studied in this section is as follows: We are 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. $\nabla f(x)$).

Motivation. Recall that the primary reason we are interested in this problem is that 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. For example, the loss could be the SVM loss function and the inputs are both the training data $(x_i, y_i), i = 1 \ldots N$ and the weights and biases W, 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. However, as we will see later in the class the gradient on x_i can still be useful sometimes, for example for purposes of visualization and interpreting what the Neural Network might be doing.

If you are coming to this class and you're comfortable with deriving gradients with chain rule, we would still like to encourage you to at least skim this section, since it presents a rarely developed view of backpropagation as backward flow in real-valued circuits and any insights you'll gain may help you throughout the class.

Simple expressions and interpretation of the gradient

Lets start simple so that we can develop the notation and conventions for more complex expressions. Consider a simple multiplication function of two numbers $f(x, y) = xy$. It is a matter of simple calculus to derive the partial derivative for either input:

$$f(x, y) = xy \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$$

Interpretation. Keep in mind what the derivatives tell you: They indicate the rate of change of a function with respect to that variable surrounding an infinitesimally small region near a particular point:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

A technical note is that the division sign on the left-hand side is, unlike the division sign on the right-hand side, not a division. Instead, this notation indicates that the operator $\frac{d}{dx}$ is being applied to the function f, and returns a different function (the derivative). A nice way to think about the expression above is that when h is very small, then the function is well-approximated by a straight line, and the derivative is its slope. In other words, the derivative on each variable tells you the sensitivity of the whole expression on its value. For example, if $x = 4, y = -3$ then

$f(x, y) = -12$ and the derivative on x $\frac{\partial f}{\partial x} = -3$. This tells us that if we were to increase the value of this variable by a tiny amount, the effect on the whole expression would be to decrease it (due to the negative sign), and by three times that amount. This can be seen by rearranging the above equation ($f(x + h) = f(x) + h\frac{df(x)}{dx}$). Analogously, since $\frac{\partial f}{\partial y} = 4$, we expect that increasing the value of y by some very small amount h would also increase the output of the function (due to the positive sign), and by $4h$.

> *The derivative on each variable tells you the sensitivity of the whole expression on its value.*

As mentioned, the gradient ∇f is the vector of partial derivatives, so we have that $\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}] = [y, x]$. Even though the gradient is technically a vector, we will often use terms such as *"the gradient on x"* instead of the technically correct phrase *"the partial derivative on x"* for simplicity.

We can also derive the derivatives for the addition operation:

$$f(x, y) = x + y \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$$

that is, the derivative on both x, y is one regardless of what the values of x, y are. This makes sense, since increasing either x, y would increase the output of f, and the rate of that increase would be independent of what the actual values of x, y are (unlike the case of multiplication above). The last function we'll use quite a bit in the class is the *max* operation:

$$f(x, y) = \max(x, y) \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = 1(x >= y) \qquad \frac{\partial f}{\partial y} = 1(y >= x)$$

That is, the (sub)gradient is 1 on the input that was larger and 0 on the other input. Intuitively, if the inputs are $x = 4, y = 2$, then the max is 4, and the function is not sensitive to the setting of y. That is, if we were to increase it by a tiny amount h, the function would keep outputting 4, and therefore the gradient is zero: there is no effect. Of course, if we were to change y by a large amount (e.g. larger than 2), then the value of f would change, but the derivatives tell us nothing about the effect of such large changes on the inputs of a function; They are only informative for tiny, infinitesimally small changes on the inputs, as indicated by the $\lim_{h \to 0}$ in its definition.

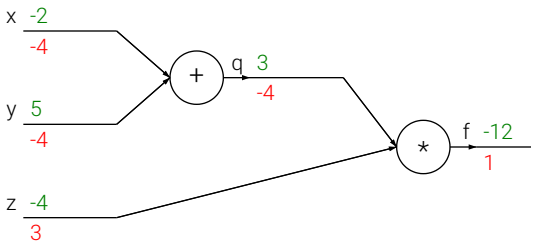
Compound expressions with chain rule

Lets now start to consider more complicated expressions that involve multiple composed functions, such as $f(x, y, z) = (x + y)z$. 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: $q = x + y$ 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 $\frac{\partial q}{\partial x} = 1$, $\frac{\partial q}{\partial y} = 1$. 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. Lets see this with an example:

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
dqdx = 1.0
dqdy = 1.0
# now backprop through q = x + y
dfdx = dfdq * dqdx  # The multiplication here is the chain rule!
dfdy = dfdq * dqdy
```

We are left with the gradient in the variables `[dfdx,dfdy,dfdz]`, which tell us the sensitivity of the variables `x,y,z` on `f`!. This is the simplest example of backpropagation. Going forward, we will use a more concise notation that omits the `df` prefix. For example, we will simply write `dq` instead of `dfdq`, and always assume that the gradient is computed on the final output.

This computation can also be nicely visualized with a circuit diagram:



The real-valued *"circuit"* on left 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.

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 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.

> *This extra multiplication (for each input) due to the chain rule can turn a single and relatively useless gate into a cog in a complex circuit such as an entire neural network.*

Lets get an intuition for how this works by referring again to the example. The add gate received inputs [-2, 5] and computed output 3. Since the gate is computing the addition operation, its local gradient for both of its inputs is +1. The rest of the circuit computed the final value, which is -12. During the backward pass in which the chain rule is applied recursively backwards through the circuit, the add gate (which is an input to the multiply gate) learns that the gradient for its output was -4. If we anthropomorphize the circuit as wanting to output a higher value (which can help with intuition), then we can think of the circuit as "wanting" the output of the add gate to be lower (due to negative sign), and with a *force* of 4. To continue the recurrence and to chain the gradient, the add gate takes that gradient and multiplies it to all of the local gradients for its inputs (making the gradient on both **x** and **y** 1 * -4 = -4). Notice that this has the desired effect: If **x,y** were to decrease (responding to their negative gradient) then the add gate's output would decrease, which in turn makes the multiply gate's output increase.

Backpropagation can thus be thought of as gates communicating to each other (through the gradient signal) whether they want their outputs to increase or decrease (and how strongly), so as to make the final output value higher.

Modularity: Sigmoid example

The gates we introduced above are relatively arbitrary. Any kind of differentiable function can act as a gate, and we can group multiple gates into a single gate, or decompose a function into multiple gates whenever it is convenient. Lets look at another expression that illustrates this point:

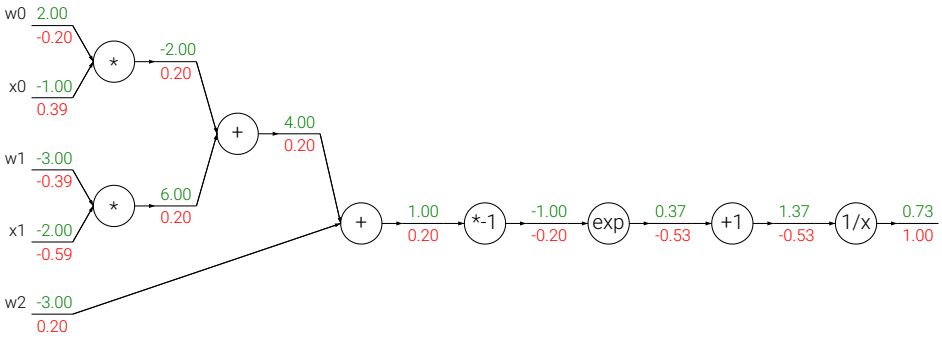
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

as we will see later in the class, this expression describes a 2-dimensional neuron (with inputs **x** and weights **w**) that uses the *sigmoid activation* function. But for now lets think of this very simply as just a function from inputs *w,x* to a single number. The function is made up of multiple gates. In addition to the ones described already above (add, mul, max), there are four more:

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2 \quad f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \quad f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} =$$

Where the functions f_c, f_a translate the input by a constant of c and scale the input by a constant of a, respectively. [illegible] technically special cases of addition and multiplication, but we introduce them as (new) unary gates

here since we do not need the gradients for the constants c, a. The full circuit then looks as follows:



Example circuit for a 2D neuron with a sigmoid activation function. The inputs are [x0,x1] and the (learnable) weights of the neuron are [w0,w1,w2]. As we will see later, the neuron computes a dot product with the input and then its activation is softly squashed by the sigmoid function to be in range from 0 to 1.

In the example above, we see a long chain of function applications that operates on the result of the dot product between **w,x**. The function that these operations implement is called the *sigmoid function* $\sigma(x)$. It turns out that the derivative of the sigmoid function with respect to its input simplifies if you perform the derivation (after a fun tricky part where we add and subtract a 1 in the numerator):

$$\sigma(x) = \frac{1}{1+e^{-x}} \rightarrow \quad \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

As we see, the gradient turns out to simplify and becomes surprisingly simple. For example, the sigmoid expression receives the input 1.0 and computes the output 0.73 during the forward pass. The derivation above shows that the *local* gradient would simply be (1 - 0.73) * 0.73 ~= 0.2, as the circuit computed before (see the image above), except this way it would be done with a single, simple and efficient expression (and with less numerical issues). Therefore, in any real practical application it would be very useful to group these operations into a single gate. Lets see the backprop for this neuron in code:

```
w = [2,-3,-3] # assume some random weights and data
x = [-1, -2]

# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function

# backward pass through the neuron (backpropagation)
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient derivation
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
# we're done! we have the gradients on the inputs to the circuit
```

Implementation protip: staged backpropagation. As shown in the code above, in practice it is always helpful to break down the forward pass into stages that are easily backpropped through. For example here we created an intermediate variable `dot` which holds the output of the dot product between `w` and `x`. During backward pass we then successively compute (in reverse order) the corresponding variables (e.g. `ddot`, and ultimately `dw, dx`) that hold the gradients of those variables.

The point of this section is that the details of how the backpropagation is performed, and which parts of the forward function we think of as gates, is a matter of convenience. It helps to be aware of which parts of the expression have easy local gradients, so that they can be chained together with the least amount of code and effort.

Backprop in practice: Staged computation

Lets see this with another example. Suppose that we have a function of the form:

$$f(x,y) = \frac{x+\sigma(y)}{\sigma(x)+(x+y)^2}$$

To be clear, this function is completely useless and it's not clear why you would ever want to compute its gradient, except for the fact that it is a good example of backpropagation in practice. It is very important to stress that if you were to launch into performing the differentiation with respect to either x or y, you would end up with very large and complex expressions. However, it turns out that doing so is completely unnecessary because we don't need to have an explicit function written down that evaluates the gradient. We only have to know how to compute it. Here is how we would structure the forward pass of such expression:

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                               #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
den = sigx + xpysqr # denominator                        #(6)
invden = 1.0 / den                                       #(7)
f = num * invden # done!                                 #(8)
```

Phew, by the end of the expression we have computed the forward pass. Notice that we have structured the code in such way that it contains multiple intermediate variables, each of which are only simple expressions for which we already know the local gradients. Therefore, computing the backprop pass is easy: We'll go backwards and for every variable along the way in the forward pass (`sigy, num, sigx, xpy, xpysqr, den, invden`) we will have the same variable, but one that begins with a `d`, which will hold the gradient of the output of the circuit with respect to that variable. Additionally, note that every single piece in our backprop will involve computing the local gradient of that expression, and chaining it with the gradient on that expression with a multiplication. For each row, we also highlight which part of the forward pass it refers to:

```
# backprop f = num * invden
dnum = invden # gradient on numerator                             #(8)
dinvden = num                                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                                #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                                #(6)
dxpysqr = (1) * dden                                              #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                        #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                                   #(4)
dy = (1) * dxpy                                                   #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                                  #(2)
dsigy = (1) * dnum                                                #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                                 #(1)
# done! phew
```

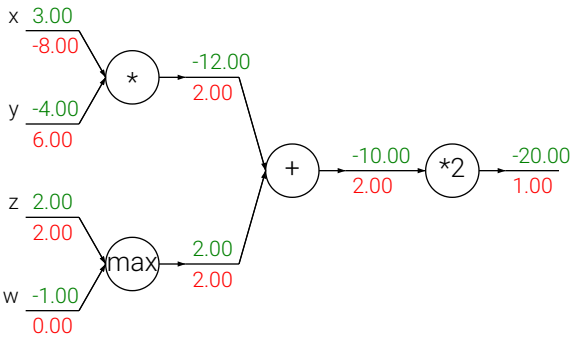
Notice a few things:

Cache forward pass variables. To compute the backward pass it is very helpful to have some of the variables that were used in the forward pass. In practice you want to structure your code so that you cache these variables, and so that they are available during backpropagation. If this is too difficult, it is possible (but wasteful) to recompute them.

Gradients add up at forks. The forward expression involves the variables **x,y** multiple times, so when we perform backpropagation we must be careful to use `+=` instead of `=` to accumulate the gradient on these variables (otherwise we would overwrite it). This follows the *multivariable chain rule* in Calculus, which states that if a variable branches out to different parts of the circuit, then the gradients that flow back to it will add.

Patterns in backward flow

It is interesting to note that in many cases the backward-flowing gradient can be interpreted on an intuitive level. For example, the three most commonly used gates in neural networks (*add,mul,max*), all have very simple interpretations in terms of how they act during backpropagation. Consider this example circuit:



An example circuit demonstrating the intuition behind the operations that backpropagation performs during the backward pass in order to compute the gradients on the inputs. Sum operation distributes gradients equally to all its inputs. Max operation routes the gradient to the higher input. Multiply gate takes the input activations, swaps them and multiplies by its gradient.

Looking at the diagram above as an example, we can see that:

The **add 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. This follows from the fact that the local gradient for the add operation is simply +1.0, so the gradients on all inputs will exactly equal the gradients on the output because it will be multiplied by x1.0 (and remain unchanged). In the example circuit above, note that the + gate routed the gradient of 2.00 to both of its inputs, equally and unchanged.

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 (unchanged) to exactly one of its inputs (the input that had the highest value during the forward pass). This is because the local gradient for a max gate is 1.0 for the highest value, and 0.0 for all other values. In the example circuit above, the max operation routed the gradient of 2.00 to the **z** variable, which had a higher value than **w**, and the gradient on **w** remains zero.

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. In the example above, the gradient on **x** is -8.00, which is -4.00 x 2.00.

Unintuitive effects and their consequences. Notice that if one of the inputs to the multiply gate is very small and the other is very big, then the multiply gate will do something slightly unintuitive: it will assign a relatively huge gradient to the small input and a tiny gradient to the large input. Note that in linear classifiers where the weights are dot producted $w^T x_i$ (multiplied) with the inputs, this implies that the scale of the data has an effect on the magnitude of the gradient for the weights. For example, if you multiplied all input data examples x_i by 1000 during preprocessing, then the gradient on the weights will be 1000 times larger, and you'd have to lower the learning rate by that factor to compensate. This is why preprocessing matters a lot, sometimes in subtle ways! And having intuitive understanding for how the gradients flow can help you debug some of these cases.

Gradients for vectorized operations

The above sections were concerned with single variables, but all concepts extend in a straight-forward manner to matrix and vector operations. However, one must pay closer attention to dimensions and transpose operations.

Matrix-Matrix multiply gradient. Possibly the most tricky operation is the matrix-matrix multiplication (which generalizes all matrix-vector and vector-vector) multiply operations:

```
# forward pass
W = np.random.randn(5, 10)
X = np.random.randn(10, 3)
D = W.dot(X)

# now suppose we had the gradient on D from above in the circuit
dD = np.random.randn(*D.shape) # same shape as D
dW = dD.dot(X.T) #.T gives the transpose of the matrix
dX = W.T.dot(dD)
```

Tip: use dimension analysis! Note that you do not need to remember the expressions for `dW` and `dX` because they are easy to re-derive based on dimensions. For instance, we know that the gradient on the weights `dW` must

be of the same size as `W` after it is computed, and that it must depend on matrix multiplication of `X` and `dD` (as is the case when both `X,W` are single numbers and not matrices). There is always exactly one way of achieving this so that the dimensions work out. For example, `X` is of size [10 x 3] and `dD` of size [5 x 3], so if we want `dW` and `W` has shape [5 x 10], then the only way of achieving this is with `dD.dot(X.T)`, as shown above.

Work with small, explicit examples. Some people may find it difficult at first to derive the gradient updates for some vectorized expressions. Our recommendation is to explicitly write out a minimal vectorized example, derive the gradient on paper and then generalize the pattern to its efficient, vectorized form.

Erik Learned-Miller has also written up a longer related document on taking matrix/vector derivatives which you might find helpful. Find it here.

Summary

- We developed intuition for what the gradients mean, how they flow backwards in the circuit, and how they communicate which part of the circuit should increase or decrease and with what force to make the final output higher.
- We discussed the importance of **staged computation** for practical implementations of backpropagation. You always want to break up your function into modules for which you can easily derive local gradients, and then chain them with chain rule. Crucially, you almost never want to write out these expressions on paper and differentiate them symbolically in full, because you never need an explicit mathematical equation for the gradient of the input variables. Hence, decompose your expressions into stages such that you can differentiate every stage independently (the stages will be matrix vector multiplies, or max operations, or sum operations, etc.) and then backprop through the variables one step at a time.

In the next section we will start to define neural networks, and backpropagation will allow us to efficiently compute the gradient of a loss function with respect to its parameters. In other words, we're now ready to train neural nets, and the most conceptually difficult part of this class is behind us! ConvNets will then be a small step away.

References

- Automatic differentiation in machine learning: a survey

Table of Contents:

Quick intro

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 $s = W_2 \max(0, W_1 x)$. Here, W_1 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 we'll 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).

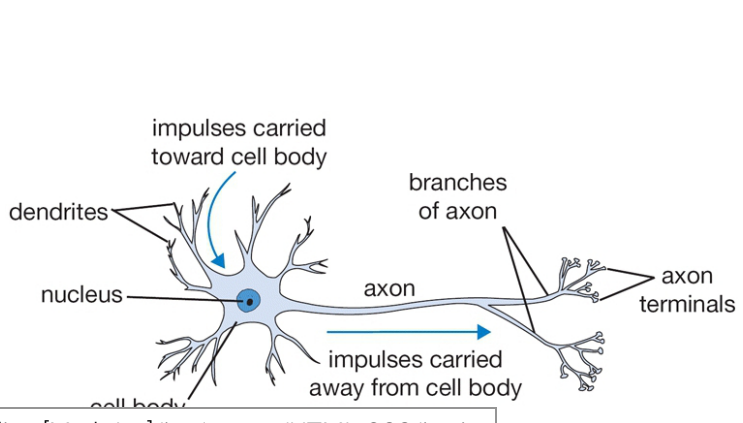
A three-layer neural network could analogously look like $s = W_3 \max(0, W_2 \max(0, W_1 x))$, where all of W_3, W_2, W_1 are parameters to be learned. The sizes of the intermediate hidden vectors are hyperparameters of the network and we'll see how we can set them later. Lets now look into how we can interpret these computations from the neuron/network perspective.

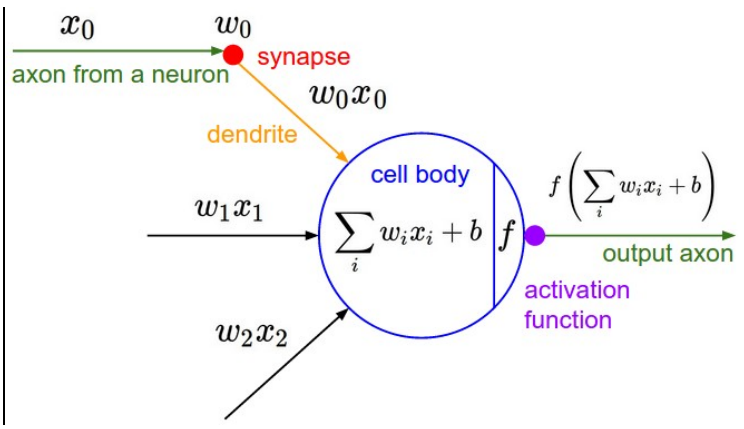
Modeling one neuron

The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems, but has since diverged and become a matter of engineering and achieving good results in Machine Learning tasks. Nonetheless, we begin our discussion with a very brief and high-level description of the biological system that a large portion of this area has been inspired by.

Biological motivation and connections

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**. The diagram below shows a cartoon drawing of a biological neuron (left) and a common mathematical model (right). 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. $w_0 x_0$) 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: excitory (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. In the computational model, we assume that the precise timings of the spikes do not matter, and that only the frequency of the firing communicates information. 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** σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0 and 1. We will see details of these activation functions later in this section.





A cartoon drawing of a biological neuron (left) and its mathematical model (right).

An example code for forward-propagating a single neuron might look as follows:

```
class Neuron(object):
  # ...
  def forward(self, inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation funct
    return firing_rate
```

In other words, each neuron performs a dot product with the input and its weights, adds the bias and applies the non-linearity (or activation function), in this case the sigmoid $\sigma(x) = 1/(1+e^{-x})$. We will go into more details about different activation functions at the end of this section.

Coarse model. It's important to stress that this model of a biological neuron is very coarse: For example, there are many different types of neurons, each with different properties. The dendrites in biological neurons perform complex nonlinear computations. The synapses are not just a single weight, they're a complex non-linear dynamical system. The exact timing of the output spikes in many systems is known to be important, suggesting that the rate code approximation may not hold. Due to all these and many other simplifications, be prepared to hear groaning sounds from anyone with some neuroscience background if you draw analogies between Neural Networks and real brains. See this review (pdf), or more recently this review if you are interested.

Single neuron as a linear classifier

The mathematical form of the model Neuron's 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 neuron's output, we can turn a single neuron into a linear classifier:

Binary Softmax classifier. For example, we can interpret $\sigma(\sum_i w_i x_i + b)$ to be the probability of one of the classes $P(y_i = 1 \mid x_i; w)$. The probability of the other class would be $P(y_i = 0 \mid x_i; w) = 1 - P(y_i = 1 \mid x_i; w)$, 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.

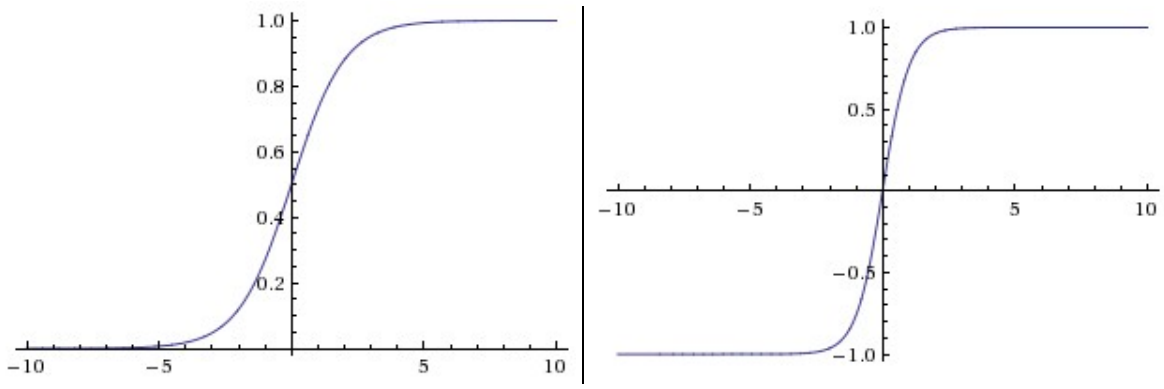
Binary SVM classifier. Alternatively, we could attach a max-margin hinge loss to the output of the neuron and train it to become a binary Support Vector Machine.

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.

> *A single neuron can be used to implement a binary classifier (e.g. binary Softmax or binary SVM classifiers)*

Commonly used activation functions

Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions you may encounter in practice:



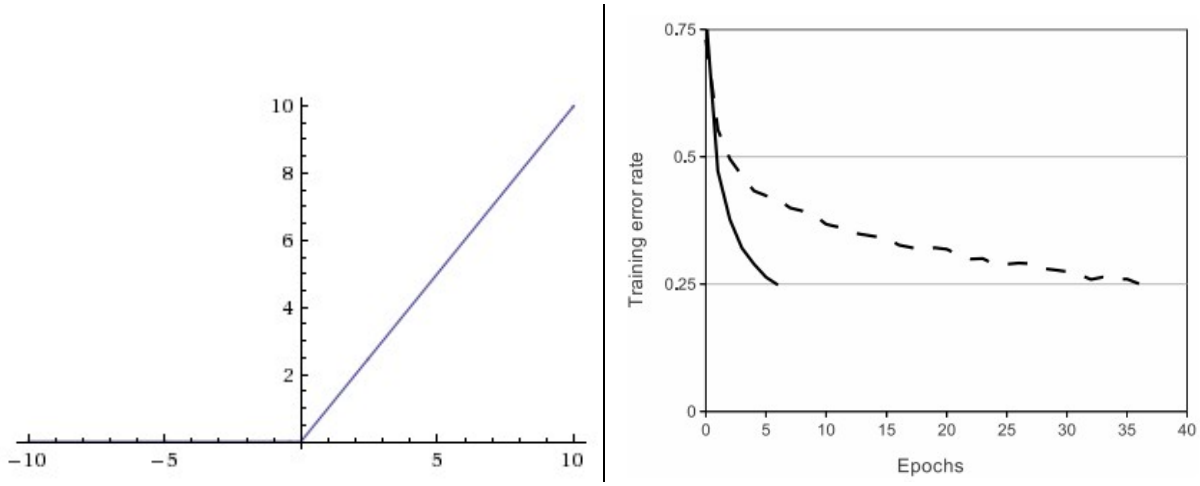
Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].

Sigmoid. The sigmoid non-linearity has the mathematical form $\sigma(x) = 1/(1+e^{-x})$ 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. The sigmoid function has seen frequent use historically since it 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). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:

- *Sigmoids saturate and kill gradients*. A very undesirable property of the sigmoid neuron is that when the neuron's 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 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. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.
- *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. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

Tanh. The tanh non-linearity is shown on the image above on the right. 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 nonlinearity*. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x) = 2\sigma(2x) - 1$.



Left: Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0. **Right:** A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

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 thresholded at zero (see image above on the left). There are several pros and cons to using the ReLUs:

- (+) 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.
- (+) 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. For example, you may find that as much as 40% of your

network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

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 positive slope (of 0.01, or so). That is, the function computes $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$ where α 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, introduced in Delving Deep into Rectifiers, by Kaiming He et al., 2015. However, the consistency of the benefit across tasks is presently unclear.

Maxout. Other types of units have been proposed that do not have the functional form $f(w^T x + b)$ where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the Maxout neuron (introduced recently by Goodfellow et al.) that generalizes the ReLU and its leaky version. 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.

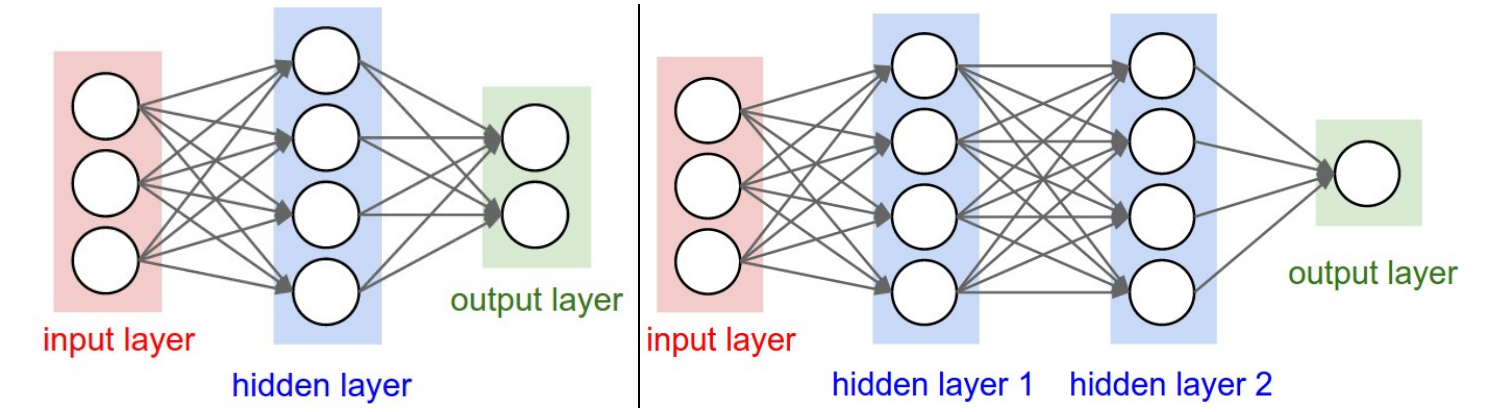
This concludes our discussion of the most common types of neurons and their activation functions. As a last comment, it is very rare to mix and match different types of neurons in the same network, even though there is no fundamental problem with doing so.

TLDR: "*What neuron type should I use?*" Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of "dead" units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

Neural Network architectures

Layer-wise organization

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. 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 (synapses) between neurons across layers, but not within a layer.

Naming conventions. Notice that when we say N-layer neural network, we do not count the input layer. Therefore, a single-layer neural network describes a network with no hidden layers (input directly mapped to output). In that sense, you can sometimes hear people say that logistic regression or SVMs are simply a special case of single-layer Neural Networks. You may also hear these networks interchangeably referred to as *"Artificial Neural Networks"* (ANN) or *"Multi-Layer Perceptrons"* (MLP). Many people do not like the analogies between Neural Networks and real brains and prefer to refer to neurons as *units*.

Output layer. Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).

Sizing neural networks. The two metrics that people commonly use to measure the size of neural networks are the number of neurons, or more commonly the number of parameters. Working with the two example networks in the above picture:

- The first network (left) has 4 + 2 = 6 neurons (not counting the inputs), [3 x 4] + [4 x 2] = 20 weights and 4 + 2 = 6 biases, for a total of 26 learnable parameters.
- The second network (right) has 4 + 4 + 1 = 9 neurons, [3 x 4] + [4 x 4] + [4 x 1] = 12 + 16 + 4 = 32 weights and 4 + 4 + 1 = 9 biases, for a total of 41 learnable parameters.

To give you some context, modern Convolutional Networks contain on orders of 100 million parameters and are usually made up of approximately 10-20 layers (hence *deep learning*). However, as we will see the number of *effective* connections is significantly greater due to parameter sharing. More on this in the Convolutional Neural Networks module.

Example feed-forward computation

Repeated matrix multiplications interwoven with activation function. One of the primary reasons that Neural

s that this structure makes it very simple and efficient to evaluate Neural

Networks using matrix vector operations. Working with the example three-layer neural network in the diagram above, the input would be a [3x1] vector. All connection strengths for a layer can be stored in a single matrix. For example, the first hidden layer's weights `W1` would be of size [4x3], and the biases for all units would be in the vector `b1`, of size [4x1]. Here, every single neuron has its weights in a row of `W1`, so the matrix vector multiplication `np.dot(W1,x)` evaluates the activations of all neurons in that layer. Similarly, `W2` would be a [4x4] matrix that stores the connections of the second hidden layer, and `W3` a [1x4] matrix for the last (output) layer. The full forward pass of this 3-layer neural network is then simply three matrix multiplications, interwoven with the application of the activation function:

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

In the above code, `W1,W2,W3,b1,b2,b3` are the learnable parameters of the network. Notice also that instead of having a single input column vector, the variable `x` could hold an entire batch of training data (where each input example would be a column of `x`) and then all examples would be efficiently evaluated in parallel. Notice that the final Neural Network layer usually doesn't have an activation function (e.g. it represents a (real-valued) class score in a classification setting).

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

Representational power

One way to look at Neural Networks with fully-connected layers is that they define a family of functions that are parameterized by the weights of the network. A natural question that arises is: What is the representational power of this family of functions? In particular, are there functions that cannot be modeled with a Neural Network?

It turns out that Neural Networks with at least one hidden layer are *universal approximators*. That is, it can be shown (e.g. see *Approximation by Superpositions of Sigmoidal Function* from 1989 (pdf), or this intuitive explanation from Michael Nielsen) that given any continuous function $f(x)$ and some $\epsilon > 0$, there exists a Neural Network $g(x)$ with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, \; | f(x) - g(x) | < \epsilon$. In other words, the neural network can approximate any continuous function.

If one hidden layer suffices to approximate any function, why use more layers and go deeper? The answer is that the fact that a two-layer Neural Network is a universal approximator is, while mathematically cute, a relatively weak and useless statement in practice. In one dimension, the "sum of indicator bumps" function $g(x) = \sum_i c_i 1(a_i < x < b_i)$ where a, b, c are parameter vectors is also a universal approximator, but noone

would suggest that we use this functional form in Machine Learning. Neural Networks work well in practice because they compactly express nice, smooth functions that fit well with the statistical properties of data we encounter in practice, and are also easy to learn using our optimization algorithms (e.g. gradient descent). Similarly, the fact that deeper networks (with multiple hidden layers) can work better than a single-hidden-layer networks is an empirical observation, despite the fact that their representational power is equal.

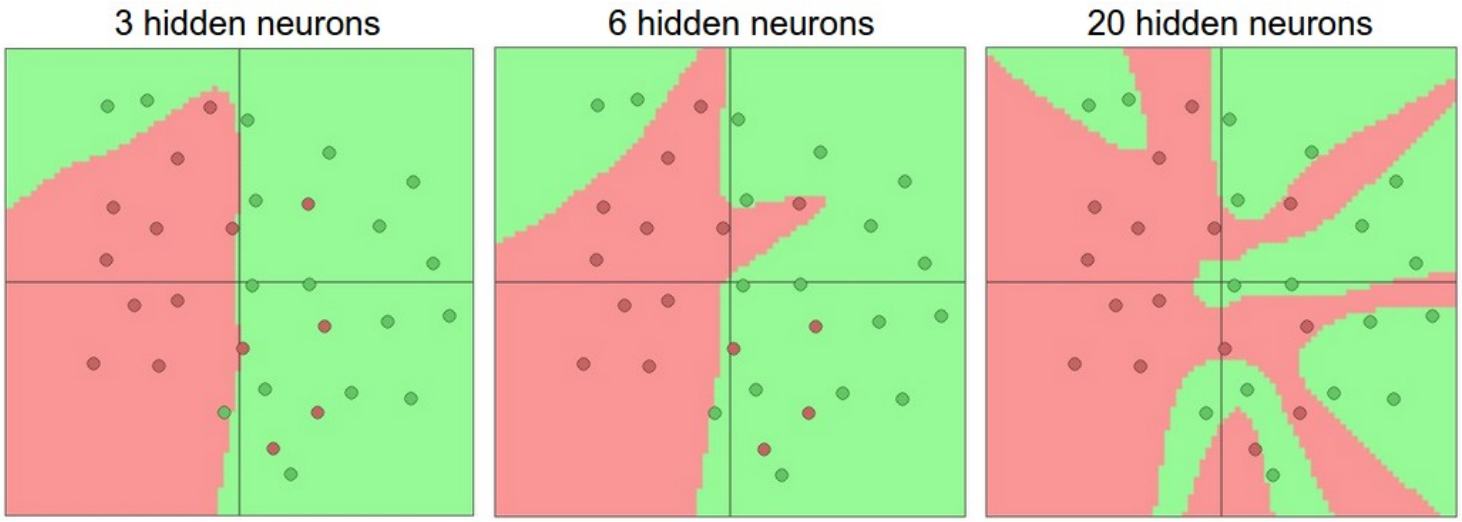
As an aside, in practice it is often the case that 3-layer neural networks will outperform 2-layer nets, but going even deeper (4,5,6-layer) rarely helps much more. This is in stark contrast to Convolutional Networks, where depth has been found to be an extremely important component for a good recognition system (e.g. on order of 10 learnable layers). One argument for this observation is that images contain hierarchical structure (e.g. faces are made up of eyes, which are made up of edges, etc.), so several layers of processing make intuitive sense for this data domain.

The full story is, of course, much more involved and a topic of much recent research. If you are interested in these topics we recommend for further reading:

- Deep Learning book in press by Bengio, Goodfellow, Courville, in particular Chapter 6.4.
- Do Deep Nets Really Need to be Deep?
- FitNets: Hints for Thin Deep Nets

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. For example, suppose we had a binary classification problem in two dimensions. We could train three separate neural networks, each with one hidden layer of some size and obtain the following classifiers:



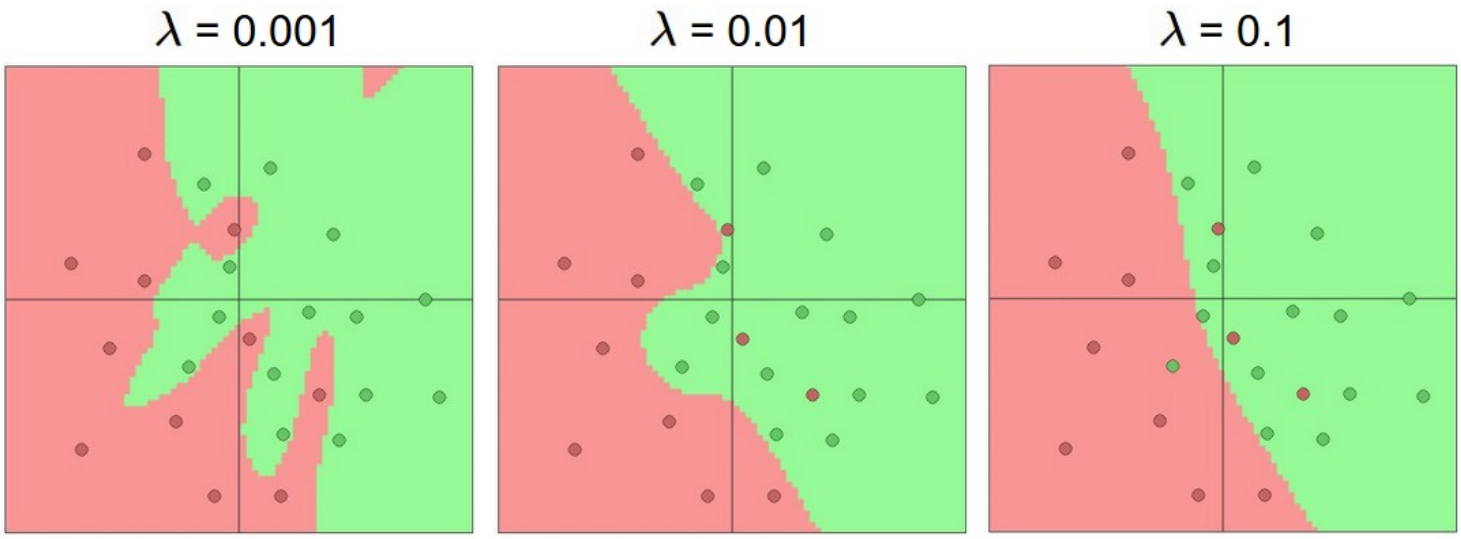
Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath. You can play with these examples in this ConvNetsJS demo.

In the diagram above, we can see that Neural Networks with more neurons can express more complicated 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. For example, the model with 20 hidden neurons fits all the training data but at the cost of segmenting the space into many disjoint red and green decision regions. The model with 3 hidden neurons only has the representational power to classify the data in broad strokes. It models the data as two blobs and interprets the few red points inside the green cluster as **outliers** (noise). In practice, this could lead to better **generalization** on the test set.

Based on our discussion above, it seems that smaller neural networks can be preferred if the data is not complex enough to prevent overfitting. However, this is incorrect - 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.

The subtle reason behind this is that smaller networks are harder to train with local methods such as Gradient Descent: It's clear that their loss functions have relatively few local minima, but it turns out that many of these minima are easier to converge to, and that they are bad (i.e. with high loss). Conversely, bigger neural networks contain significantly more local minima, but these minima turn out to be much better in terms of their actual loss. Since Neural Networks are non-convex, it is hard to study these properties mathematically, but some attempts to understand these objective functions have been made, e.g. in a recent paper The Loss Surfaces of Multilayer Networks. In practice, what you find is that if you train a small network the final loss can display a good amount of variance - in some cases you get lucky and converge to a good place but in some cases you get trapped in one of the bad minima. On the other hand, if you train a large network you'll start to find many different solutions, but the variance in the final achieved loss will be much smaller. In other words, all solutions are about equally as good, and rely less on the luck of random initialization.

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.

The takeaway is that you should not be using smaller networks because you are afraid of overfitting. Instead, you should use as big of a neural network as your computational budget allows, and use other regularization techniques to control overfitting.

Summary

In summary,

- We introduced a very coarse model of a biological **neuron**.
- We discussed several types of **activation functions** that are used in practice, with ReLU being the most common choice.
- We introduced **Neural Networks** where neurons are connected with **Fully-Connected layers** where neurons in adjacent layers have full pair-wise connections, but neurons within a layer are not connected.
- We saw that this layered architecture enables very efficient evaluation of Neural Networks based on matrix multiplications interwoven with the application of the activation function.
- We saw that that Neural Networks are **universal function approximators**, but we also discussed the fact that this property has little to do with their ubiquitous use. They are used because they make certain "right" assumptions about the functional forms of functions that come up in practice.
- We discussed the fact that larger networks will always work better than smaller networks, but their higher model capacity must be appropriately addressed with stronger regularization (such as higher weight decay), or they might overfit. We will see more forms of regularization (especially dropout) in later sections.

Additional References

- deeplearning.net tutorial with Theano
- ConvNetJS demos for intuitions
- Michael Nielsen's tutorials

Table of Contents:

Setting up the data and the model

In the previous section we introduced a model of a Neuron, which computes a dot product following a non-linearity, and Neural Networks that arrange neurons into layers. Together, these choices define the new form of the **score function**, which we have extended from the simple linear mapping that we have seen in the Linear Classification section. In particular, a Neural Network performs a sequence of linear mappings with interwoven non-linearities. In this section we will discuss additional design choices regarding data preprocessing, weight initialization, and loss functions.

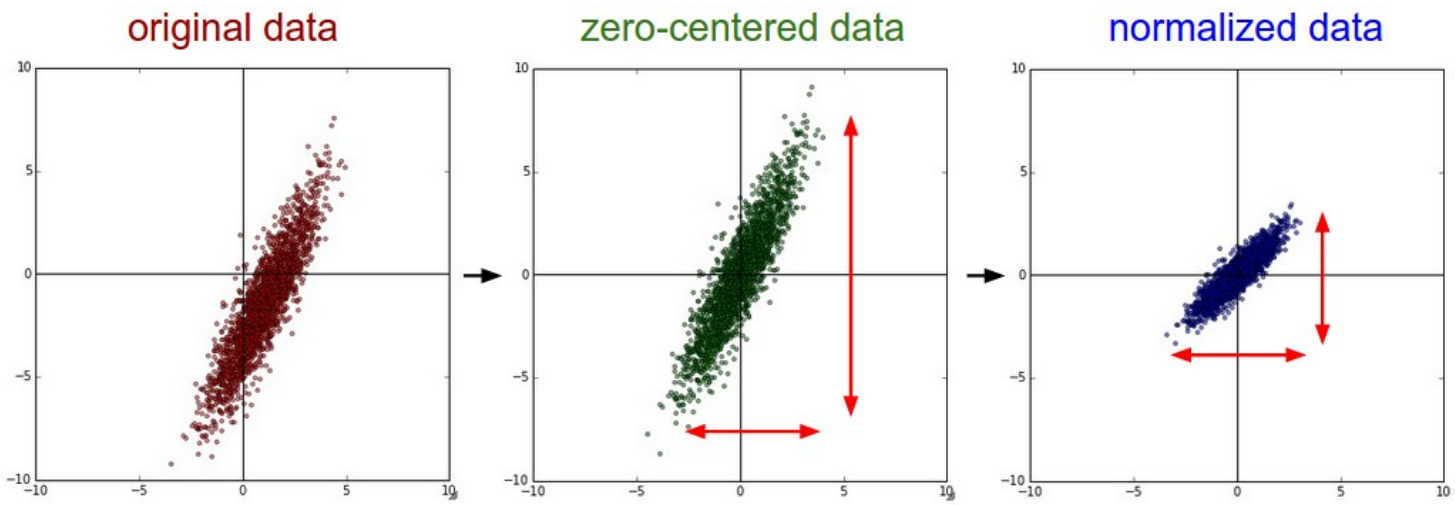
Data Preprocessing

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. In numpy, this operation would be implemented as: `X -= np.mean(X, axis = 0)`. With images specifically, for convenience it can be common to subtract a single value from all pixels (e.g. `X -= np.mean(X)`), or to do so separately across the three color channels.

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: (`X /= np.std(X, axis = 0)`). 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. In this process, the data is first centered as described above. Then, we can compute the covariance matrix that tells us about the correlation structure in the data:

```
# Assume input data matrix X of size [N x D]
X -= np.mean(X, axis = 0) # zero-center the data (important)
cov = np.dot(X.T, X) / X.shape[0] # get the data covariance matrix
```

The (i,j) element of the data covariance matrix contains the *covariance* between i-th and j-th dimension of the data. In particular, the diagonal of this matrix contains the variances. Furthermore, the covariance matrix is symmetric and positive semi-definite. We can compute the SVD factorization of the data covariance matrix:

```
U,S,V = np.linalg.svd(cov)
```

where the columns of `U` are the eigenvectors and `S` is a 1-D array of the singular values. To decorrelate the data, we project the original (but zero-centered) data into the eigenbasis:

```
Xrot = np.dot(X, U) # decorrelate the data
```

Notice that the columns of `U` are a set of orthonormal vectors (norm of 1, and orthogonal to each other), so [illegible]rded as basis vectors. The projection therefore corresponds to a rotation of the data in `X`

so that the new axes are the eigenvectors. If we were to compute the covariance matrix of `Xrot`, we would see that it is now diagonal. A nice property of `np.linalg.svd` is that in its returned value `U`, the eigenvector columns are sorted by their eigenvalues. We can use this to reduce the dimensionality of the data by only using the top few eigenvectors, and discarding the dimensions along which the data has no variance. This is also sometimes referred to as Principal Component Analysis (PCA) dimensionality reduction:

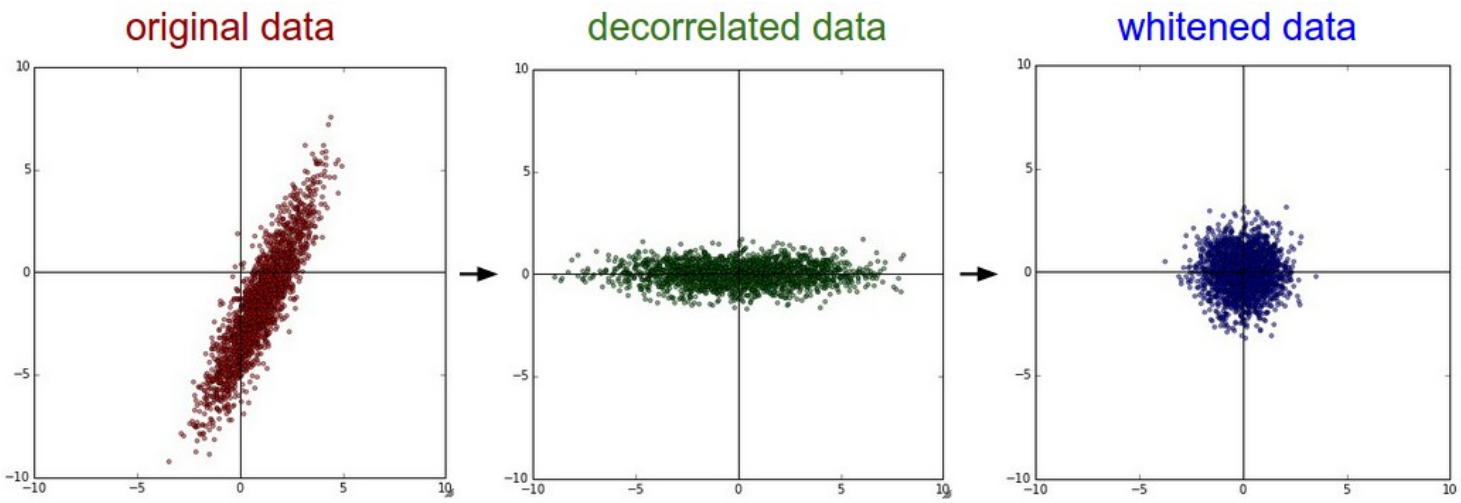
```
Xrot_reduced = np.dot(X, U[:,:100]) # Xrot_reduced becomes [N x 100]
```

After this operation, we would have reduced the original dataset of size [N x D] to one of size [N x 100], keeping the 100 dimensions of the data that contain the most variance. It is very often the case that you can get very good performance by training linear classifiers or neural networks on the PCA-reduced datasets, obtaining savings in both space and time.

The last transformation you may see in practice is **whitening**. The whitening operation takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale. The geometric interpretation of this transformation is that if the input data is a multivariable gaussian, then the whitened data will be a gaussian with zero mean and identity covariance matrix. This step would take the form:

```
# whiten the data:
# divide by the eigenvalues (which are square roots of the singular values)
Xwhite = Xrot / np.sqrt(S + 1e-5)
```

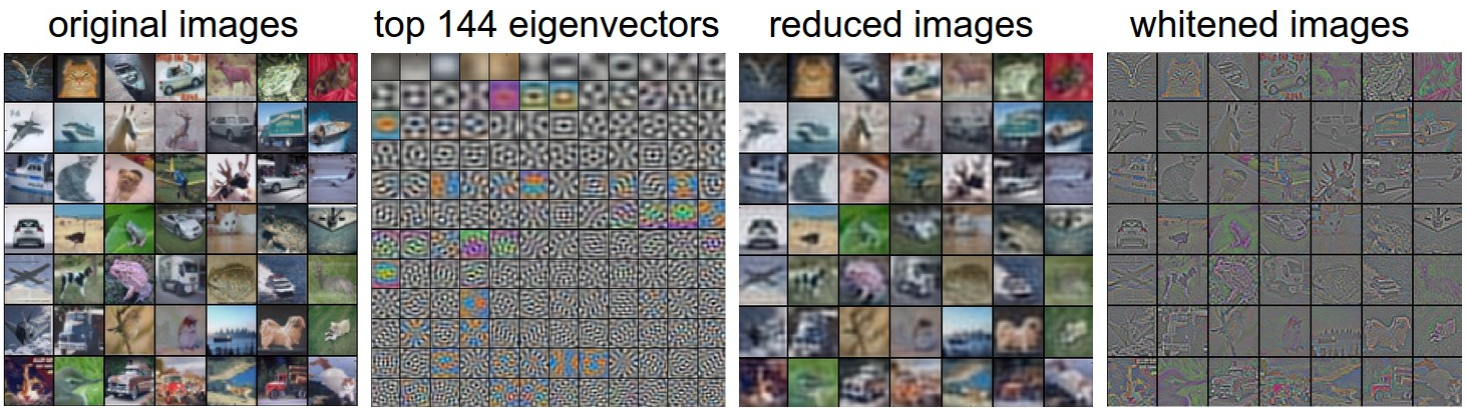
Warning: Exaggerating noise. Note that we're adding 1e-5 (or a small constant) to prevent division by zero. One weakness of this transformation is that it can greatly exaggerate the noise in the data, since it stretches all dimensions (including the irrelevant dimensions of tiny variance that are mostly noise) to be of equal size in the input. This can in practice be mitigated by stronger smoothing (i.e. increasing 1e-5 to be a larger number).



PCA / Whitening. **Left**: Original toy, 2-dimensional input data. **Middle**: After performing PCA. The data is centered at zero and then rotated into the eigenbasis of the data covariance matrix. This decorrelates the data (the covariance matrix). **Right**: Each dimension is additionally scaled by the eigenvalues, transforming the data covariance

matrix into the identity matrix. Geometrically, this corresponds to stretching and squeezing the data into an isotropic gaussian blob.

We can also try to visualize these transformations with CIFAR-10 images. The training set of CIFAR-10 is of size 50,000 x 3072, where every image is stretched out into a 3072-dimensional row vector. We can then compute the [3072 x 3072] covariance matrix and compute its SVD decomposition (which can be relatively expensive). What do the computed eigenvectors look like visually? An image might help:



Left: An example set of 49 images. **2nd from Left:** The top 144 out of 3072 eigenvectors. The top eigenvectors account for most of the variance in the data, and we can see that they correspond to lower frequencies in the images. **2nd from Right:** The 49 images reduced with PCA, using the 144 eigenvectors shown here. That is, instead of expressing every image as a 3072-dimensional vector where each element is the brightness of a particular pixel at some location and channel, every image above is only represented with a 144-dimensional vector, where each element measures how much of each eigenvector adds up to make up the image. In order to visualize what image information has been retained in the 144 numbers, we must rotate back into the "pixel" basis of 3072 numbers. Since U is a rotation, this can be achieved by multiplying by U.transpose()[:144,:], and then visualizing the resulting 3072 numbers as the image. You can see that the images are slightly blurrier, reflecting the fact that the top eigenvectors capture lower frequencies. However, most of the information is still preserved. **Right:** Visualization of the "white" representation, where the variance along every one of the 144 dimensions is squashed to equal length. Here, the whitened 144 numbers are rotated back to image pixel basis by multiplying by U.transpose()[:144,:]. The lower frequencies (which accounted for most variance) are now negligible, while the higher frequencies (which account for relatively little variance originally) become exaggerated.

In practice. We mention PCA/Whitening in these notes for completeness, but these transformations are not used with Convolutional Networks. However, it is very important to zero-center the data, and it is common to see normalization of every pixel as well.

Common pitfall. 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).

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.

Pitfall: all zero initialization. Lets start with what we should not do. Note that we do not know what the final value of every weight should be in the trained network, but with proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative. 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*. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network. The implementation for one weight matrix might look like `W = 0.01* np.random.randn(D,H)`, where `randn` samples from a zero mean, unit standard deviation gaussian. With this formulation, every neuron's weight vector is initialized as a random vector sampled from a multi-dimensional gaussian, so the neurons point in random direction in the input space. It is also possible to use small numbers drawn from a uniform distribution, but this seems to have relatively little impact on the final performance in practice.

Warning: It's 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). One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. It turns out that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its *fan-in* (i.e. its number of inputs). That is, the recommended heuristic is to initialize each neuron's weight vector as: `w = np.random.randn(n) / sqrt(n)`, where `n` is the number of its inputs. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence.

The sketch of the derivation is as follows: Consider the inner product $s = \sum_i^n w_i x_i$ between the weights w and input x, which gives the raw activation of a neuron before the non-linearity. We can examine the variance of s:

$$
\begin{aligned}
\text{Var}(s) &= \text{Var}\left(\sum_i^n w_i x_i\right) \\
&= \sum_i^n \text{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + [E(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i)\text{Var}(w_i) \\
&= \sum_i^n \text{Var}(x_i)\text{Var}(w_i) \\
&= (n\text{Var}(w))\text{Var}(x)
\end{aligned}
$$

where in the first 2 steps we have used properties of variance. In third step we assumed zero mean inputs and weights, so $E[x_i] = E[w_i] = 0$. Note that this is not generally the case: For example ReLU units will have a positive mean. In the last step we assumed that all w_i, x_i are identically distributed. From this derivation we can see that if we want s to have the same variance as all of its inputs x, then during initialization we should make sure that the variance of every weight w is $1/n$. And since $\text{Var}(aX) = a^2\text{Var}(X)$ for a random variable X and a scalar a, this implies that we should draw from unit gaussian and then scale it by $a = \sqrt{1/n}$, to make its variance $1/n$. This gives the initialization `w = np.random.randn(n) / sqrt(n)`.

A similar analysis is carried out in Understanding the difficulty of training deep feedforward neural networks by Glorot et al. In this paper, the authors end up recommending an initialization of the form $\text{Var}(w) = 2/(n_{in} + n_{out})$ where n_{in}, n_{out} are the number of units in the previous layer and the next layer. This is based on a compromise and an equivalent analysis of the backpropagated gradients. A more recent paper on this topic, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification by He et al., derives an initialization specifically for ReLU neurons, reaching the conclusion that the variance of neurons in the network should be $2.0/n$. This gives the initialization `w = np.random.randn(n) * sqrt(2.0/n)`, and 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.

In practice, the current recommendation is to use ReLU units and use the `w = np.random.randn(n) * sqrt(2.0/n)`, as discussed in He et al..

Batch Normalization. A recently developed technique by Ioffe and Szegedy called Batch Normalization alleviates a lot of headaches with properly initializing neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. The core observation is that this is possible because normalization is a simple differentiable operation. In the implementation, applying this technique usually amounts to insert the BatchNorm layer immediately after fully connected layers (or convolutional layers, as we'll soon see), and before non-linearities. We do not expand on this technique here because it is well described in the linked paper, but note that it has become a very common practice to use Batch Normalization in neural networks. In practice networks that use Batch Normalization are significantly more robust to bad initialization. Additionally, batch normalization can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable manner. Neat!

Regularization

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

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 $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. It is common to see the factor of $\frac{1}{2}$ in front because then the gradient of this term with respect to the parameter w is simply λw instead of $2\lambda w$. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. As we discussed in the Linear Classification section, due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather than some of its inputs a lot. Lastly, notice that during gradient descent parameter update, using the L2 regularization ultimately means that every weight is decayed linearly: `W += -lambda * W` towards zero.

L1 regularization is another relatively common form of regularization, where for each weight w we add the term $\lambda \mid w \mid$ to the objective. It is possible to combine the L1 regularization with the L2 regularization: $\lambda_1 \mid w \mid + \lambda_2 w^2$ (this is called Elastic net regularization). The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero). In other words, neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs. In comparison, final weight vectors from L2 regularization are usually diffuse, small numbers. In practice, if you are not concerned with explicit feature selection, L2 regularization can be expected to give superior performance over L1.

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. In practice, this corresponds to performing the parameter update as normal, and then enforcing

the constraint by clamping the weight vector $\vec{w}$ of every neuron to satisfy $\|\vec{w}\|_2 < c$. Typical values of c are on orders of 3 or 4. Some people report improvements when using this form of regularization. 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 is an extremely effective, simple and recently introduced regularization technique by Srivastava et al. in Dropout: A Simple Way to Prevent Neural Networks from Overfitting (pdf) that complements the other methods (L1, L2, maxnorm). While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise.

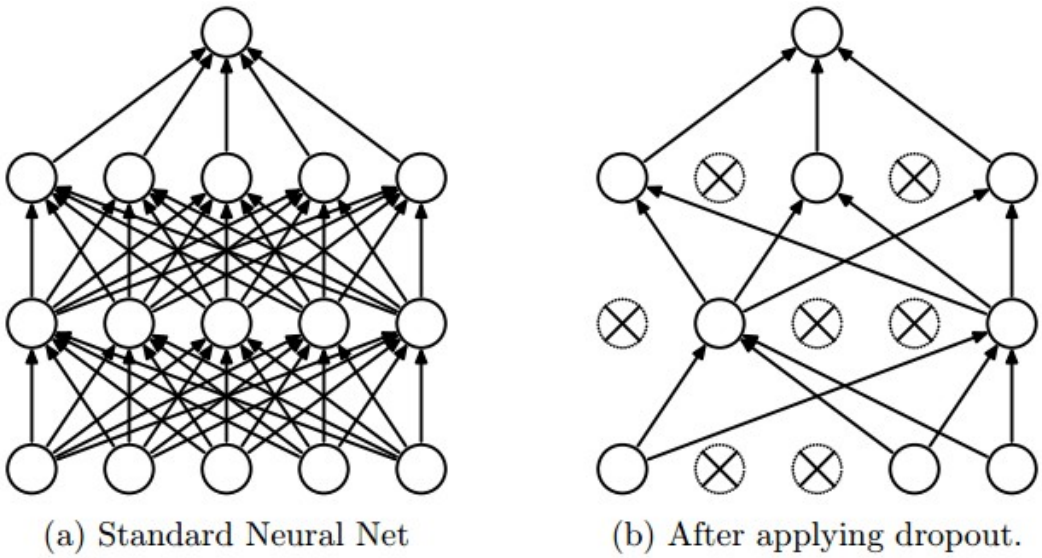


Figure taken from the Dropout paper that illustrates the idea. During training, Dropout can be interpreted as sampling a Neural Network within the full Neural Network, and only updating the parameters of the sampled network based on the input data. (However, the exponential number of possible sampled 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).

Vanilla dropout in an example 3-layer Neural Network would be implemented as follows:

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

```
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

In the code above, inside the `train_step` function we have performed dropout twice: on the first hidden layer and on the second hidden layer. It is also possible to perform dropout right on the input layer, in which case we would also create a binary mask for the input `X`. The backward pass remains unchanged, but of course has to take into account the generated masks `U1,U2`.

Crucially, note that in the `predict` function we are not dropping anymore, but we are performing a scaling of both hidden layer outputs by p. This is important because at test time all neurons see all their inputs, so we want the outputs of neurons at test time to be identical to their expected outputs at training time. For example, in case of $p = 0.5$, the neurons must halve their outputs at test time to have the same output as they had during training time (in expectation). To see this, consider an output of a neuron x (before dropout). With dropout, the expected output from this neuron will become $px + (1 - p)0$, because the neuron's output will be set to zero with probability $1 - p$. At test time, when we keep the neuron always active, we must adjust $x \rightarrow px$ to keep the same expected output. It can also be shown that performing this attenuation at test time can be related to the process of iterating over all the possible binary masks (and therefore all the exponentially many sub-networks) and computing their ensemble prediction.

The undesirable property of the scheme presented above is that we must scale the activations by p at test time. Since test-time performance is so critical, it is always preferable to use **inverted dropout**, which performs the scaling at train time, leaving the forward pass at test time untouched. Additionally, this has the appealing property that the prediction code can remain untouched when you decide to tweak where you apply dropout, or if at all. Inverted dropout looks as follows:

```
"""
Inverted Dropout: Recommended implementation example.
We drop and scale at train time and don't do anything at test time.
"""

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  andom.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
```

```
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

There has a been a large amount of research after the first introduction of dropout that tries to understand the source of its power in practice, and its relation to the other regularization techniques. Recommended further reading for an interested reader includes:

- Dropout paper by Srivastava et al. 2014.
- Dropout Training as Adaptive Regularization: "we show that the dropout regularizer is first-order equivalent to an L2 regularizer applied after scaling the features by an estimate of the inverse diagonal Fisher information matrix".

Theme of noise in forward pass. Dropout falls into a more general category of methods that introduce stochastic behavior in the forward pass of the network. During testing, the noise is marginalized over *analytically* (as is the case with dropout when multiplying by p), or *numerically* (e.g. via sampling, by performing several forward passes with different random decisions and then averaging over them). An example of other research in this direction includes DropConnect, where a random set of weights is instead set to zero during forward pass. As foreshadowing, Convolutional Neural Networks also take advantage of this theme with methods such as stochastic pooling, fractional pooling, and data augmentation. We will go into details of these methods later.

Bias regularization. As we already mentioned in the Linear Classification section, it is not common to regularize the bias parameters because they do not interact with the data through multiplicative interactions, and therefore do not have the interpretation of controlling the influence of a data dimension on the final objective. However, in practical applications (and with proper data preprocessing) regularizing the bias rarely leads to significantly worse performance. This is likely because there are very few bias terms compared to all the weights, so the classifier can "afford to" use the biases if it needs them to obtain a better data loss.

Per-layer regularization. It is not very common to regularize different layers to different amounts (except perhaps the output layer). Relatively few results regarding this idea have been published in the literature.

In practice: It is most common to use a single, global L2 regularization strength that is cross-validated. It is also common to combine this with dropout applied after all layers. The value of $p = 0.5$ is a reasonable

default, but this can be tuned on validation data.

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. The data loss takes the form of an average over the data losses for every individual example. That is, $L = \frac{1}{N}\sum_i L_i$ where N is the number of training data. Lets abbreviate $f = f(x_i; W)$ to be the activations of the output layer in a Neural Network. There are several types of problems you might want to solve in practice:

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. One of two most commonly seen cost functions in this setting is the SVM (e.g. the Weston Watkins formulation):

$$L_i = \sum_{j \neq y_i} \max\,(0, f_j - f_{y_i} + 1)$$

As we briefly alluded to, some people report better performance with the squared hinge loss (i.e. instead using $\max\,(0, f_j - f_{y_i} + 1)^2$). The second common choice is the Softmax classifier that uses the cross-entropy loss:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

Problem: Large number of classes. When the set of labels is very large (e.g. words in English dictionary, or ImageNet which contains 22,000 categories), computing the full softmax probabilities becomes expensive. For certain applications, approximate versions are popular. For instance, it may be helpful to use *Hierarchical Softmax* in natural language processing tasks (see one explanation here (pdf)). The hierarchical softmax decomposes words as labels in a tree. Each label is then represented as a path along the tree, and a Softmax classifier is trained at every node of the tree to disambiguate between the left and right branch. The structure of the tree strongly impacts the performance and is generally problem-dependent.

Attribute classification. Both losses above assume that there is a single correct answer y_i. But what if y_i is a binary vector where every example may or may not have a certain attribute, and where the attributes are not exclusive? For example, images on Instagram can be thought of as labeled with a certain subset of hashtags from a large set of all hashtags, and an image may contain multiple. A sensible approach in this case is to build a binary classifier for every single attribute independently. For example, a binary classifier for each category independently would take the form:

$$L_i = \sum_j \max(0, 1 - y_{ij} f_j)$$

where the sum is over all categories j, and y_{ij} is either +1 or -1 depending on whether the i-th example is labeled with the j-th attribute, and the score vector f_j will be positive when the class is predicted to be present and negative otherwise. Notice that loss is accumulated if a positive example has score less than +1, or when a negative example has score greater than -1.

An alternative to this loss would be to train a logistic regression classifier for every attribute independently. A binary logistic regression classifier has only two classes (0,1), and calculates the probability of class 1 as:

$$P(y = 1 \mid x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b)$$

Since the probabilities of class 1 and 0 sum to one, the probability for class 0 is $P(y = 0 \mid x; w, b) = 1 - P(y = 1 \mid x; w, b)$. Hence, an example is classified as a positive example (y = 1) if $\sigma(w^T x + b) > 0.5$, or equivalently if the score $w^T x + b > 0$. The loss function then maximizes this probability. You can convince yourself that this simplifies to minimizing the negative log-likelihood:

$$L_i = -\sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))$$

where the labels y_{ij} are assumed to be either 1 (positive) or 0 (negative), and $\sigma(\cdot)$ is the sigmoid function. The expression above can look scary but the gradient on f is in fact extremely simple and intuitive: $\partial L_i / \partial f_j = \sigma(f_j) - y_{ij}$ (as you can double check yourself by taking the derivatives).

Regression is the task of predicting real-valued quantities, such as the price of houses or the length of something in an image. For this task, it is common to compute the loss between the predicted quantity and the true answer and then measure the L2 squared norm, or L1 norm of the difference. The L2 norm squared would compute the loss for a single example of the form:

$$L_i = \|f - y_i\|_2^2$$

The reason the L2 norm is squared in the objective is that the gradient becomes much simpler, without changing the optimal parameters since squaring is a monotonic operation. The L1 norm would be formulated by summing the absolute value along each dimension:

$$L_i = \|f - y_i\|_1 = \sum_j |f_j - (y_i)_j|$$

where the sum $\sum_j$ is a sum over all dimensions of the desired prediction, if there is more than one quantity being predicted. Looking at only the j-th dimension of the i-th example and denoting the difference between the true and the predicted value by δ_{ij}, the gradient for this dimension (i.e. $\partial L_i / \partial f_j$) is easily derived to be either δ_{ij} with the L2 norm, or $sign(\delta_{ij})$. That is, the gradient on the score will either be directly proportional to the error, or it will be fixed and only inherit the sign of the difference.

Word of caution: It is important to note that the L2 loss is much harder to optimize than a more stable loss such as Softmax. Intuitively, it requires a very fragile and specific property from the network to output exactly one correct value for each input (and its augmentations). Notice that this is not the case with Softmax, where the precise value of each score is less important: It only matters that their magnitudes are appropriate. Additionally, the L2 loss is less robust because outliers can introduce huge gradients. When faced with a regression problem, first consider if it is absolutely inadequate to quantize the output into bins. For example, if you are predicting star rating for a product, it might work much better to use 5 independent classifiers for ratings of 1-5 stars instead of a regression loss. Classification has the additional benefit that it can give you a distribution over the regression outputs, not just a single output with no indication of its confidence. If you're certain that classification is not appropriate, use the L2 but be careful: For example, the L2 is more fragile and applying dropout in the network (especially in the layer right before the L2 loss) is not a great idea.

> *When faced with a regression task, first consider if it is absolutely necessary. Instead, have a strong preference to discretizing your outputs to bins and perform classification over them whenever possible.*

Structured prediction. The structured loss refers to a case where the labels can be arbitrary structures such as graphs, trees, or other complex objects. Usually it is also assumed that the space of structures is very large and not easily enumerable. The basic idea behind the structured SVM loss is to demand a margin between the correct structure y_i and the highest-scoring incorrect structure. It is not common to solve this problem as a simple unconstrained optimization problem with gradient descent. Instead, special solvers are usually devised so that the specific simplifying assumptions of the structure space can be taken advantage of. We mention the problem briefly but consider the specifics to be outside of the scope of the class.

Summary

In summary:

- The recommended preprocessing is to center the data to have mean of zero, and normalize its scale to [-1, 1] along each feature
- Initialize the weights by drawing them from a gaussian distribution with standard deviation of $\sqrt{2/n}$, where n is the number of inputs to the neuron. E.g. in numpy: `w = np.random.randn(n) * sqrt(2.0/n)`.
- Use L2 regularization and dropout (the inverted version)
- Use batch normalization
- We discussed different tasks you might want to perform in practice, and the most common loss functions for each task

We've now preprocessed the data and set up and initialized the model. In the next section we will look at the learning process and its dynamics.

Table of Contents:

Learning

In the previous sections we've 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.

Gradient Checks

In theory, performing a gradient check is as simple as comparing the analytic gradient to the numerical gradient. In practice, the process is much more involved and error prone. Here are some tips, tricks, and issues to watch out for:

Use the centered formula. The formula you may have seen for the finite difference approximation when evaluating the numerical gradient looks as follows:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

where h is a very small number, in practice approximately 1e-5 or so. In practice, it turns out that it is much better to use the *centered* difference formula of the form:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

This requires you to evaluate the loss function twice to check every single dimension of the gradient (so it is about 2 times as expensive), but the gradient approximation turns out to be much more precise. To see this, you can use Taylor expansion of $f(x+h)$ and $f(x-h)$ and verify that the first formula has an error on order of $O(h)$, while the second formula only has error terms on order of $O(h^2)$ (i.e. it is a second order approximation).

Use relative error for the comparison. What are the details of comparing the numerical gradient f_n' and analytic gradient f_a'? That is, how do we know if the two are not compatible? You might be temped to keep track of the difference $|f_a' - f_n'|$ or its square and define the gradient check as failed if that difference is above a threshold. However, this is problematic. For example, consider the case where their difference is 1e-4. This seems like a very appropriate difference if the two gradients are about 1.0, so we'd consider the two gradients to match. But if the gradients were both on order of 1e-5 or lower, then we'd consider 1e-4 to be a huge difference and likely a failure. Hence, it is always more appropriate to consider the *relative error*:

$$\frac{|f_a' - f_n'|}{\max(|f_a'|, |f_n'|)}$$

which considers their ratio of the differences to the ratio of the absolute values of both gradients. Notice that normally the relative error formula only includes one of the two terms (either one), but I prefer to max (or add) both to make it symmetric and to prevent dividing by zero in the case where one of the two is zero (which can often happen, especially with ReLUs). However, one must explicitly keep track of the case where both are zero and pass the gradient check in that edge case. In practice:

- relative error > 1e-2 usually means the gradient is probably wrong
- 1e-2 > relative error > 1e-4 should make you feel uncomfortable
- 1e-4 > relative error is usually okay for objectives with kinks. But if there are no kinks (e.g. use of tanh nonlinearities and softmax), then 1e-4 is too high.
- 1e-7 and less you should be happy.

Also keep in mind that the deeper the network, the higher the relative errors will be. So if you are gradient checking the input data for a 10-layer network, a relative error of 1e-2 might be okay because the errors build up on the way. Conversely, an error of 1e-2 for a single differentiable function likely indicates incorrect gradient.

Use double precision. A common pitfall is using single precision floating point to compute gradient check. It is often that case that you might get high relative errors (as high as 1e-2) even with a correct gradient implementation. In my experience I've sometimes seen my relative errors plummet from 1e-2 to 1e-8 by switching to double precision.

Stick around active range of floating point. It's a good idea to read through "What Every Computer Scientist Should Know About Floating-Point Arithmetic", as it may demystify your errors and enable you to write more careful code. For example, in neural nets it can be common to normalize the loss function over the batch. However, if your gradients per datapoint are very small, then *additionally* dividing them by the number of data points is starting to give very small numbers, which in turn will lead to more numerical issues. This is why I like to always print the raw numerical/analytic gradient, and make sure that the numbers you are comparing are not extremely small (e.g. roughly 1e-10 and smaller in absolute value is worrying). If they are you may want to temporarily scale your loss function up by a constant to bring them to a "nicer" range where floats are more dense - ideally on the order of 1.0, where your float exponent is 0.

Kinks in the objective. One source of inaccuracy to be aware of during gradient checking is the problem of *kinks*. Kinks refer to non-differentiable parts of an objective function, introduced by functions such as ReLU ($\max(0, x)$), or the SVM loss, Maxout neurons, etc. Consider gradient checking the ReLU function at $x = -1e6$. Since $x < 0$, the analytic gradient at this point is exactly zero. However, the numerical gradient would suddenly compute a non-zero gradient because $f(x + h)$ might cross over the kink (e.g. if $h > 1e-6$) and introduce a non-zero contribution. You might think that this is a pathological case, but in fact this case can be very common. For example, an SVM for CIFAR-10 contains up to 450,000 $\max(0, x)$ terms because there are 50,000 examples and each example yields 9 terms to the objective. Moreover, a Neural Network with an SVM classifier will contain many more kinks due to ReLUs.

Note that it is possible to know if a kink was crossed in the evaluation of the loss. This can be done by keeping track of the identities of all "winners" in a function of form $\max(x, y)$; That is, was x or y higher during the forward pass. If the identity of at least one winner changes when evaluating $f(x + h)$ and then $f(x - h)$, then a kink was crossed and the numerical gradient will not be exact.

Use only few datapoints. One fix to the above problem of kinks is to use fewer datapoints, since loss functions that contain kinks (e.g. due to use of ReLUs or margin losses etc.) will have fewer kinks with fewer datapoints, so it is less likely for you to cross one when you perform the finite different approximation. Moreover, if your gradcheck for only ~2 or 3 datapoints then you would almost certainly gradcheck for an entire batch. Using very few datapoints also makes your gradient check faster and more efficient.

Be careful with the step size h. It is not necessarily the case that smaller is better, because when h is much smaller, you may start running into numerical precision problems. Sometimes when the gradient doesn't check, it is possible that you change h to be 1e-4 or 1e-6 and suddenly the gradient will be correct. This wikipedia article contains a chart that plots the value of **h** on the x-axis and the numerical gradient error on the y-axis.

Gradcheck during a "characteristic" mode of operation. It is important to realize that a gradient check is performed at a particular (and usually random), single point in the space of parameters. Even if the gradient check succeeds at that point, it is not immediately certain that the gradient is correctly implemented globally.

Additionally, a random initialization might not be the most "characteristic" point in the space of parameters and may in fact introduce pathological situations where the gradient seems to be correctly implemented but isn't. For instance, an SVM with very small weight initialization will assign almost exactly zero scores to all datapoints and the gradients will exhibit a particular pattern across all datapoints. An incorrect implementation of the gradient could still produce this pattern and not generalize to a more characteristic mode of operation where some scores are larger than others. Therefore, to be safe it is best to use a short **burn-in** time during which the network is allowed to learn and perform the gradient check after the loss starts to go down. The danger of performing it at the first iteration is that this could introduce pathological edge cases and mask an incorrect implementation of the gradient.

Don't let the regularization overwhelm the data. It is often the case that a loss function is a sum of the data loss and the regularization loss (e.g. L2 penalty on weights). One danger to be aware of is that the regularization loss may overwhelm the data loss, in which case the gradients will be primarily coming from the regularization term (which usually has a much simpler gradient expression). This can mask an incorrect implementation of the data loss gradient. Therefore, it is recommended to turn off regularization and check the data loss alone first, and then the regularization term second and independently. One way to perform the latter is to hack the code to remove the data loss contribution. Another way is to increase the regularization strength so as to ensure that its effect is non-negligible in the gradient check, and that an incorrect implementation would be spotted.

Remember to turn off dropout/augmentations. When performing gradient check, remember to turn off any non-deterministic effects in the network, such as dropout, random data augmentations, etc. Otherwise these can clearly introduce huge errors when estimating the numerical gradient. The downside of turning off these effects is that you wouldn't be gradient checking them (e.g. it might be that dropout isn't backpropagated correctly). Therefore, a better solution might be to force a particular random seed before evaluating both $f(x + h)$ and $f(x - h)$, and when evaluating the analytic gradient.

Check only few dimensions. In practice the gradients can have sizes of million parameters. In these cases it is only practical to check some of the dimensions of the gradient and assume that the others are correct. **Be careful**: One issue to be careful with is to make sure to gradient check a few dimensions for every separate parameter. In some applications, people combine the parameters into a single large parameter vector for convenience. In these cases, for example, the biases could only take up a tiny number of parameters from the whole vector, so it is important to not sample at random but to take this into account and check that all parameters receive the correct gradients.

Before learning: sanity checks Tips/Tricks

Here are a few sanity checks you might consider running before you plunge into expensive optimization:

- **Look for correct loss at chance performance.** Make sure you're getting the loss you expect when you initialize with small parameters. It's best to first check the data loss alone (so set regularization strength to zero). For example, for CIFAR-10 with a Softmax classifier we would expect the initial loss to be 2.302, because we expect a diffuse probability of 0.1 for each class (since there are 10 classes), and Softmax loss is the negative log probability of the correct class so: -ln(0.1) = 2.302. For The

Weston Watkins SVM, we expect all desired margins to be violated (since all scores are approximately zero), and hence expect a loss of 9 (since margin is 1 for each wrong class). If you're not seeing these losses there might be issue with initialization.

- As a second sanity check, increasing the regularization strength should increase the loss
- **Overfit a tiny subset of data**. Lastly and most importantly, before training on the full dataset try to train on a tiny portion (e.g. 20 examples) of your data and make sure you can achieve zero cost. For this experiment it's also best to set regularization to zero, otherwise this can prevent you from getting zero cost. Unless you pass this sanity check with a small dataset it is not worth proceeding to the full dataset. Note that it may happen that you can overfit very small dataset but still have an incorrect implementation. For instance, if your datapoints' features are random due to some bug, then it will be possible to overfit your small training set but you will never notice any generalization when you fold it your full dataset.

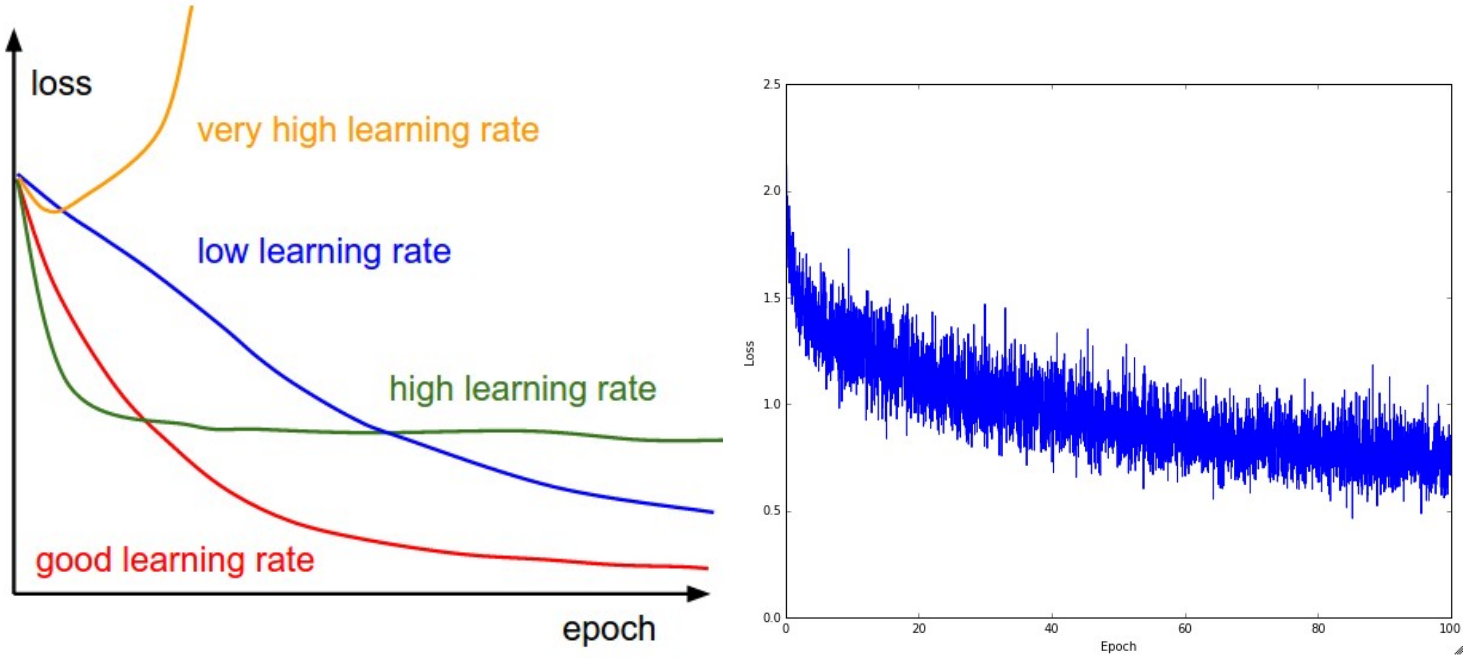
Babysitting the learning process

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.

The x-axis of the plots below are always in units of epochs, which measure how many times every example has been seen during training in expectation (e.g. one epoch means that every example has been seen once). It is preferable to track epochs rather than iterations since the number of iterations depends on the arbitrary setting of batch size.

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. Below is a cartoon diagram showing the loss over time, and especially what the shape might tell you about the learning rate:



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).

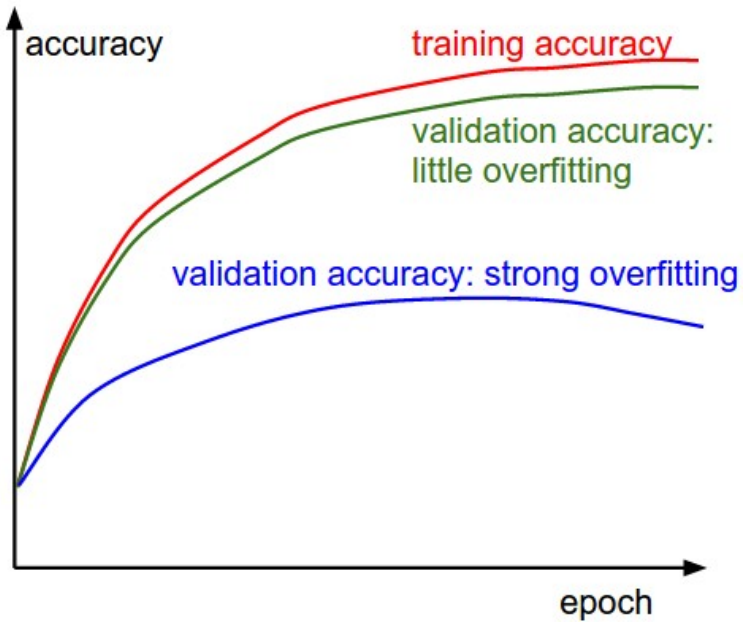
Some people prefer to plot their loss functions in the log domain. Since learning progress generally takes an exponential form shape, the plot appears as a slightly more interpretable straight line, rather than a hockey stick. Additionally, if multiple cross-validated models are plotted on the same loss graph, the differences between them become more apparent.

Sometimes loss functions can look funny lossfunctions.tumblr.com.

Train/Val accuracy

The second important quantity to track while training a classifier is the validation/training accuracy. This plot can give you valuable insights into the amount of overfitting in your model:

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.

Ratio of weights:updates

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. Here is a specific example:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

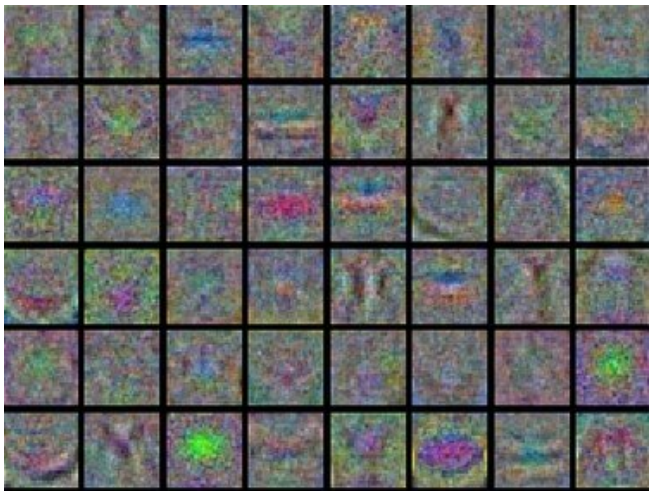
Instead of tracking the min or the max, some people prefer to compute and track the norm of the gradients and their updates instead. These metrics are usually correlated and often give approximately the same results.

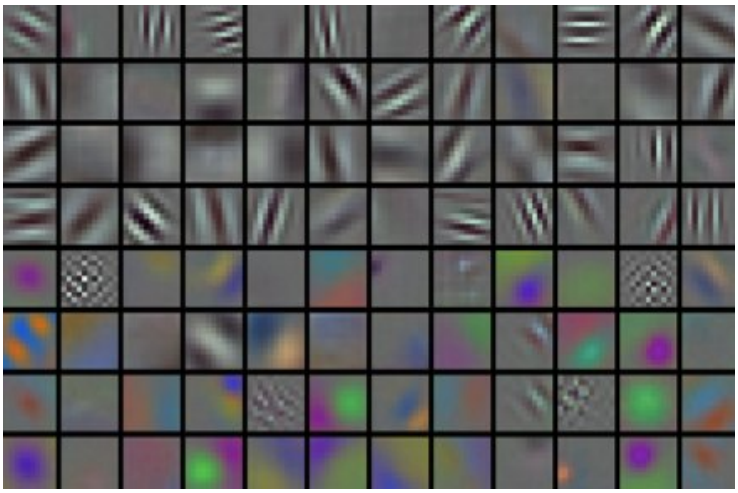
Activation / Gradient distributions per layer

An incorrect initialization can slow down or even completely stall the learning process. Luckily, this issue can be diagnosed relatively easily. One way to do so is to plot activation/gradient histograms for all layers of the network. Intuitively, it is not a good sign to see any strange distributions - e.g. with tanh neurons we would like to see a distribution of neuron activations between the full range of [-1,1], instead of seeing all neurons outputting zero, or all neurons being completely saturated at either -1 or 1.

First-layer Visualizations

Lastly, when one is working with image pixels it can be helpful and satisfying to plot the first-layer features visually:





Examples of visualized weights for the first layer of a neural network. **Left**: Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. **Right:** Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

Parameter updates

Once the analytic gradient is computed with backpropagation, the gradients are used to perform a parameter update. There are several approaches for performing the update, which we discuss next.

We note that optimization for deep networks is currently a very active area of research. In this section we highlight some established and common techniques you may see in practice, briefly describe their intuition, but leave a detailed analysis outside of the scope of the class. We provide some further pointers for an interested reader.

SGD and bells and whistles

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
```

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 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 $\mathrm{U} = \mathrm{mgh}$ and therefore $\mathrm{U} \propto \mathrm{h}$). 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.

Since the force on the particle is related to the gradient of potential energy (i.e. $\mathrm{F} = -\nabla \mathrm{U}$), the **force** felt by the particle is precisely the (negative) **gradient** of the loss function. Moreover, $\mathrm{F} = \mathrm{ma}$ so the (negative) gradient is in this view proportional to the acceleration of the particle. Note that this is different from the SGD update shown above, where the gradient directly integrates the position. Instead, the physics view suggests an update in which the gradient only directly influences the velocity, which in turn has an effect on the position:

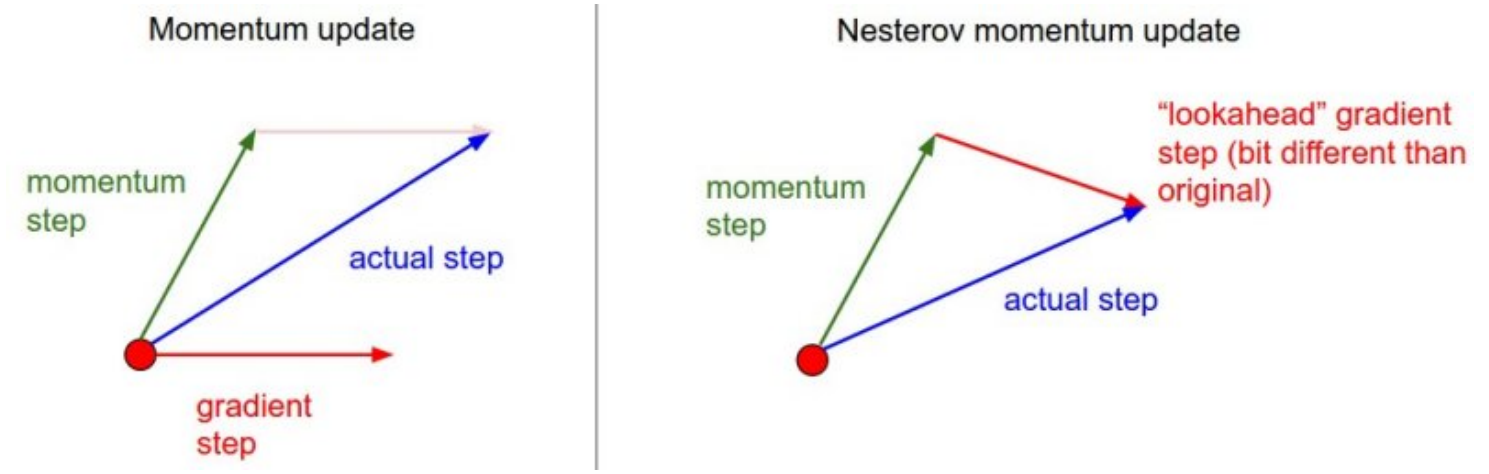
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Here we see an introduction of a `v` variable that is initialized at zero, and an additional hyperparameter (`mu`). As an unfortunate misnomer, this variable is in optimization referred to as *momentum* (its typical value is about 0.9), but 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. When cross-validated, this parameter is usually set to values such as [0.5, 0.9, 0.95, 0.99]. Similar to annealing schedules for learning rates (discussed later, below), optimization can sometimes benefit a little from momentum schedules, where the momentum is increased in later stages of learning. A typical setting is to start with momentum of about 0.5 and anneal it to 0.99 or so over multiple epochs.

> *With Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient.*

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.

The core idea behind Nesterov momentum is that when the current parameter vector is at some position `x`, then looking at the momentum update above, we know that the momentum term alone (i.e. ignoring the second term with the gradient) is about to nudge the parameter vector by `mu * v`. Therefore, if we are about to compute the gradient, we can treat the future approximate position `x + mu * v` as a "lookahead" - this is a point in the vicinity of where we are soon going to end up. Hence, it makes sense to compute the gradient at `x + mu * v` instead of at the "old/stale" position `x`.



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.

That is, in a slightly awkward notation, we would like to do the following:

```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

However, in practice people prefer to express the update to look as similar to vanilla SGD or to the previous momentum update as possible. This is possible to achieve by manipulating the update above with a variable transform `x_ahead = x + mu * v`, and then expressing the update in terms of `x_ahead` instead of `x`. That is, the parameter vector we are actually storing is always the ahead version. The equations in terms of `x_ahead` (but renaming it back to `x`) then become:

```
v_prev = v # back this up
v = mu * v - learning_rate * dx # velocity update stays the same
x += -mu * v_prev + (1 + mu) * v # position update changes form
```

We recommend this further reading to understand the source of these equations and the mathematical formulation of Nesterov's Accelerated Momentum (NAG):

- Advances in optimizing Recurrent Networks by Yoshua Bengio, Section 3.5.
- Ilya Sutskever's thesis (pdf) contains a longer exposition of the topic in section 7.2

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.

Knowing when to decay the learning rate can be tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. There are three common types of implementing the learning rate decay:

- **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 $\alpha = \alpha_0 e^{-kt}$, where α_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 $\alpha = \alpha_0/(1 + kt)$ where a_0, k are hyperparameters and t is the iteration number.

In practice, we find that the step decay is slightly preferable because the hyperparameters it involves (the fraction of decay and the step timings in units of epochs) are more interpretable than the hyperparameter k. Lastly, if you can afford the computational budget, err on the side of slower decay and train for a longer time.

Second order methods

A second, popular group of methods for optimization in context of deep learning is based on Newton's method, which iterates the following update:

$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

Here, $Hf(x)$ is the Hessian matrix, which is a square matrix of second-order partial derivatives of the function. The term $\nabla f(x)$ is the gradient vector, as seen in Gradient Descent. Intuitively, the Hessian describes the local curvature of the loss function, which allows us to perform a more efficient update. In particular, multiplying by the inverse Hessian leads the optimization to take more aggressive steps in directions of shallow curvature and shorter steps in directions of steep curvature. Note, crucially, the absence of any learning rate hyperparameters in the update formula, which the proponents of these methods cite this as a large advantage over first-order methods.

However, the update above is impractical for most deep learning applications because computing (and inverting) the Hessian in its explicit form is a very costly process in both space and time. For instance, a Neural Network with one million parameters would have a Hessian matrix of size [1,000,000 x 1,000,000], occupying approximately 3725 gigabytes of RAM. Hence, a large variety of *quasi-Newton* methods have been developed that seek to approximate the inverse Hessian. Among these, the most popular is L-BFGS, which uses the information in the gradients over time to form the approximation implicitly (i.e. the full matrix is never computed).

However, even after we eliminate the memory concerns, a large downside of a naive application of L-BFGS is that it must be computed over the entire training set, which could contain millions of examples. Unlike mini-

batch SGD, getting L-BFGS to work on mini-batches is more tricky and an active area of research.

In practice, it is currently not common to see L-BFGS or similar second-order methods applied to large-scale Deep Learning and Convolutional Neural Networks. Instead, SGD variants based on (Nesterov's) momentum are more standard because they are simpler and scale more easily.

Additional references:

- Large Scale Distributed Deep Networks is a paper from the Google Brain team, comparing L-BFGS and SGD variants in large-scale distributed optimization.
- SFO algorithm strives to combine the advantages of SGD with advantages of L-BFGS.

Per-parameter adaptive learning rate methods

All previous approaches we've 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. Many of these methods may still require other hyperparameter settings, but the argument is that they are well-behaved for a broader range of hyperparameter values than the raw learning rate. In this section we highlight some common adaptive methods you may encounter in practice:

Adagrad is an adaptive learning rate method originally proposed by Duchi et al..

```
# Assume the gradient dx and parameter vector x
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. Notice that 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. Amusingly, the square root operation turns out to be very important and without it the algorithm performs much worse. 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.

RMSprop. RMSprop is a very effective, but currently unpublished adaptive learning rate method. Amusingly, everyone who uses this method in their work currently cites slide 29 of Lecture 6 of Geoff Hinton's Coursera class. 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]. Notice that the `x+=` update is identical to Adagrad, but the `cache` variable is a "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.

Adam. 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)
```

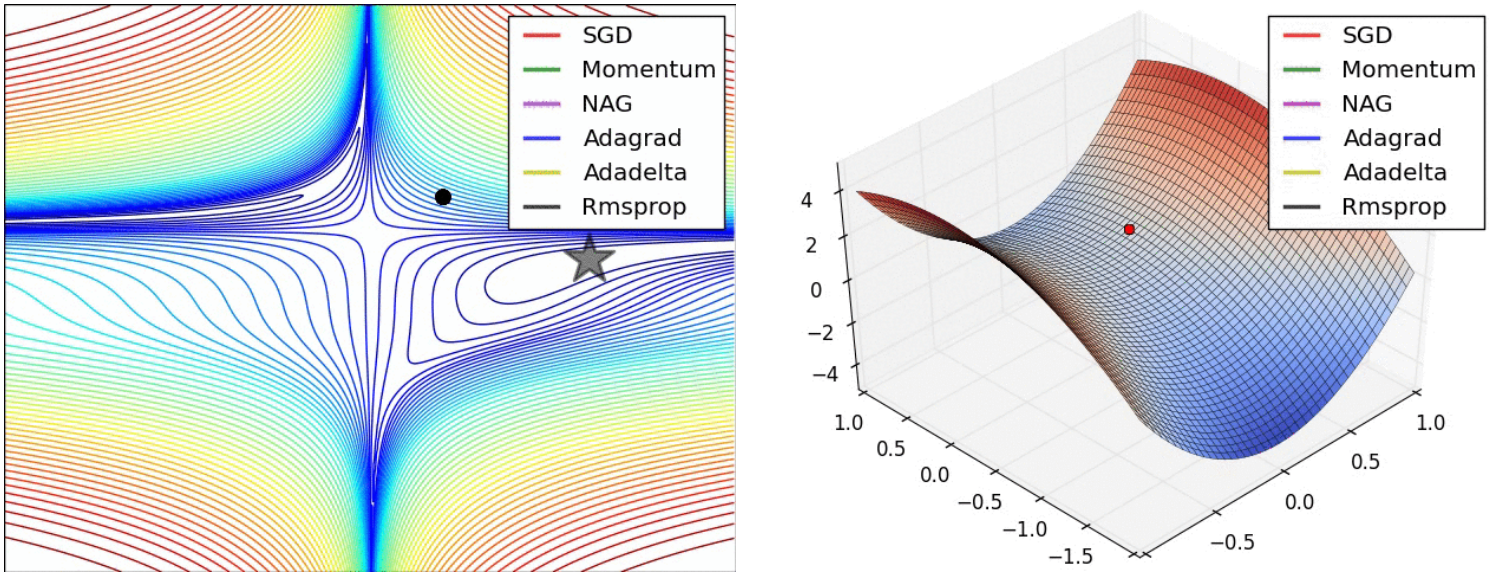
Notice that the update looks exactly as RMSProp update, except the "smooth" version of the gradient `m` is used instead of the raw (and perhaps noisy) gradient vector `dx`. Recommended values in the paper are `eps = 1e-8`, `beta1 = 0.9`, `beta2 = 0.999`. In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. However, it is often also worth trying SGD+Nesterov Momentum as an alternative. 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 your 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)
```

Note that the update is now a function of the iteration as well as the other parameters. We refer the reader to the paper for the details, or the course slides where this is expanded on.

Additional References:

- Unit Tests for Stochastic Optimization proposes a series of tests as a standardized benchmark for stochastic optimization.



Animations that may help your intuitions about the learning process dynamics. **Left:** Contours of a loss surface and time evolution of different optimization algorithms. Notice the "overshooting" behavior of momentum-based methods, which make the optimization look like a ball rolling down the hill. **Right:** A visualization of a saddle point in the optimization landscape, where the curvature along different dimension has different signs (one dimension curves up and another down). Notice that SGD has a very hard time breaking symmetry and gets stuck on the top. Conversely, algorithms such as RMSprop will see very low gradients in the saddle direction. Due to the denominator term in the RMSprop update, this will increase the effective learning rate along this direction, helping RMSProp proceed. Images credit: Alec Radford.

Hyperparameter optimization

As we've 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)
- regularization strength (L2 penalty, dropout strength)

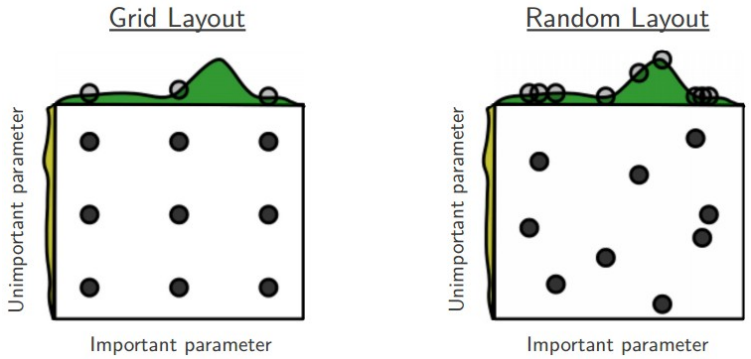
But as we saw, there are many more relatively less sensitive hyperparameters, for example in per-parameter adaptive learning methods, the setting of momentum and its schedule, etc. In this section we describe some additional tips and tricks for performing the hyperparameter search:

Implementation. Larger Neural Networks typically require a long time to train, so performing hyperparameter search can take many days/weeks. It is important to keep this in mind since it influences the design of your code base. One particular design is to have a **worker** that continuously samples random hyperparameters and performs the optimization. During the training, the worker will keep track of the validation performance after every epoch, and writes a model checkpoint (together with miscellaneous training statistics such as the loss over time) to a file, preferably on a shared file system. It is useful to include the validation performance directly in the filename, so that it is simple to inspect and sort the progress. Then there is a second program which we will call a **master**, which launches or kills workers across a computing cluster, and may additionally inspect the checkpoints written by workers and plot their training statistics, etc.

Prefer one validation fold to cross-validation. In most cases a single validation set of respectable size substantially simplifies the code base, without the need for cross-validation with multiple folds. You'll hear people say they "cross-validated" a parameter, but many times it is assumed that they still only used a single validation set.

Hyperparameter ranges. Search for hyperparameters on log scale. For example, a typical sampling of the learning rate would look as follows: `learning_rate = 10 ** uniform(-6, 1)`. That is, we are generating a random number from a uniform distribution, but then raising it to the power of 10. The same strategy should be used for the regularization strength. Intuitively, this is because learning rate and regularization strength have multiplicative effects on the training dynamics. For example, a fixed change of adding 0.01 to a learning rate has huge effects on the dynamics if the learning rate is 0.001, but nearly no effect if the learning rate when it is 10. This is because the learning rate multiplies the computed gradient in the update. Therefore, it is much more natural to consider a range of learning rate multiplied or divided by some value, than a range of learning rate added or subtracted to by some value. Some parameters (e.g. dropout) are instead usually searched in the original scale (e.g. `dropout = uniform(0,1)`).

Prefer random search to grid search. As argued by Bergstra and Bengio in Random Search for Hyper-Parameter Optimization, "randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid". As it turns out, this is also usually easier to implement.



Core illustration from Random Search for Hyper-Parameter Optimization by Bergstra and Bengio. It is very often the case that some of the hyperparameters matter much more than others (e.g. top hyperparam vs. left one in this figure). Performing random search rather than grid search allows you to much more precisely discover good values for the important ones.

Careful with best values on border. Sometimes it can happen that you're searching for a hyperparameter (e.g. learning rate) in a bad range. For example, suppose we use `learning_rate = 10 ** uniform(-6, 1)`. Once we receive the results, it is important to double check that the final learning rate is not at the edge of this interval, or otherwise you may be missing more optimal hyperparameter setting beyond the interval.

Stage your search from coarse to fine. In practice, it can be helpful to first search in coarse ranges (e.g. 10 ** [-6, 1]), and then depending on where the best results are turning up, narrow the range. Also, it can be helpful to perform the initial coarse search while only training for 1 epoch or even less, because many hyperparameter settings can lead the model to not learn at all, or immediately explode with infinite cost. The

second stage could then perform a narrower search with 5 epochs, and the last stage could perform a detailed search in the final range for many more epochs (for example).

Bayesian Hyperparameter Optimization is a whole area of research devoted to coming up with algorithms that try to more efficiently navigate the space of hyperparameters. The core idea is to appropriately balance the exploration - exploitation trade-off when querying the performance at different hyperparameters. Multiple libraries have been developed based on these models as well, among some of the better known ones are Spearmint, SMAC, and Hyperopt. However, in practical settings with ConvNets it is still relatively difficult to beat random search in a carefully-chosen intervals. See some additional from-the-trenches discussion here.

Evaluation

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. There are a few approaches to forming an ensemble:

- **Same model, different initializations**. Use cross-validation to determine the best hyperparameters, then train multiple models with the best set of hyperparameters but with different random initialization. The danger with this approach is that the variety is only due to initialization.
- **Top models discovered during cross-validation**. Use cross-validation to determine the best hyperparameters, then pick the top few (e.g. 10) models to form the ensemble. This improves the variety of the ensemble but has the danger of including suboptimal models. In practice, this can be easier to perform since it doesn't require additional retraining of models after cross-validation
- **Different checkpoints of a single model**. If training is very expensive, some people have had limited success in taking different checkpoints of a single network over time (for example after every epoch) and using those to form an ensemble. Clearly, this suffers from some lack of variety, but can still work reasonably well in practice. The advantage of this approach is that is very cheap.
- **Running average of parameters during training**. Related to the last point, a cheap way of almost always getting an extra percent or two of performance is to maintain a second copy of the network's weights in memory that maintains an exponentially decaying sum of previous weights during training. This way you're averaging the state of the network over last several iterations. You will find that this "smoothed" version of the weights over last few steps almost always achieves better validation error. The rough intuition to have in mind is that the objective is bowl-shaped and your network is jumping around the mode, so the average has a higher chance of being somewhere nearer the mode.

One disadvantage of model ensembles is that they take longer to evaluate on test example. An interested reader may find the recent work from Geoff Hinton on "Dark Knowledge" inspiring, where the idea is to "distill"

a good ensemble back to a single model by incorporating the ensemble log likelihoods into a modified objective.

Summary

To train a Neural Network:

- Gradient check your implementation with a small batch of data and be aware of the pitfalls.
- As a sanity check, make sure your initial loss is reasonable, and that you can achieve 100% training accuracy on a very small portion of the data
- During training, monitor the loss, the training/validation accuracy, and if you're feeling fancier, the magnitude of updates in relation to parameter values (it should be ~1e-3), and when dealing with ConvNets, the first-layer weights.
- The two recommended updates to use are either SGD+Nesterov Momentum or Adam.
- Decay your learning rate over the period of the training. For example, halve the learning rate after a fixed number of epochs, or whenever the validation accuracy tops off.
- Search for good hyperparameters with random search (not grid search). Stage your search from coarse (wide hyperparameter ranges, training only for 1-5 epochs), to fine (narrower rangers, training for many more epochs)
- Form model ensembles for extra performance

Additional References

- SGD tips and tricks from Leon Bottou
- Efficient BackProp (pdf) from Yann LeCun
- Practical Recommendations for Gradient-Based Training of Deep Architectures from Yoshua Bengio

---

Table of Contents:

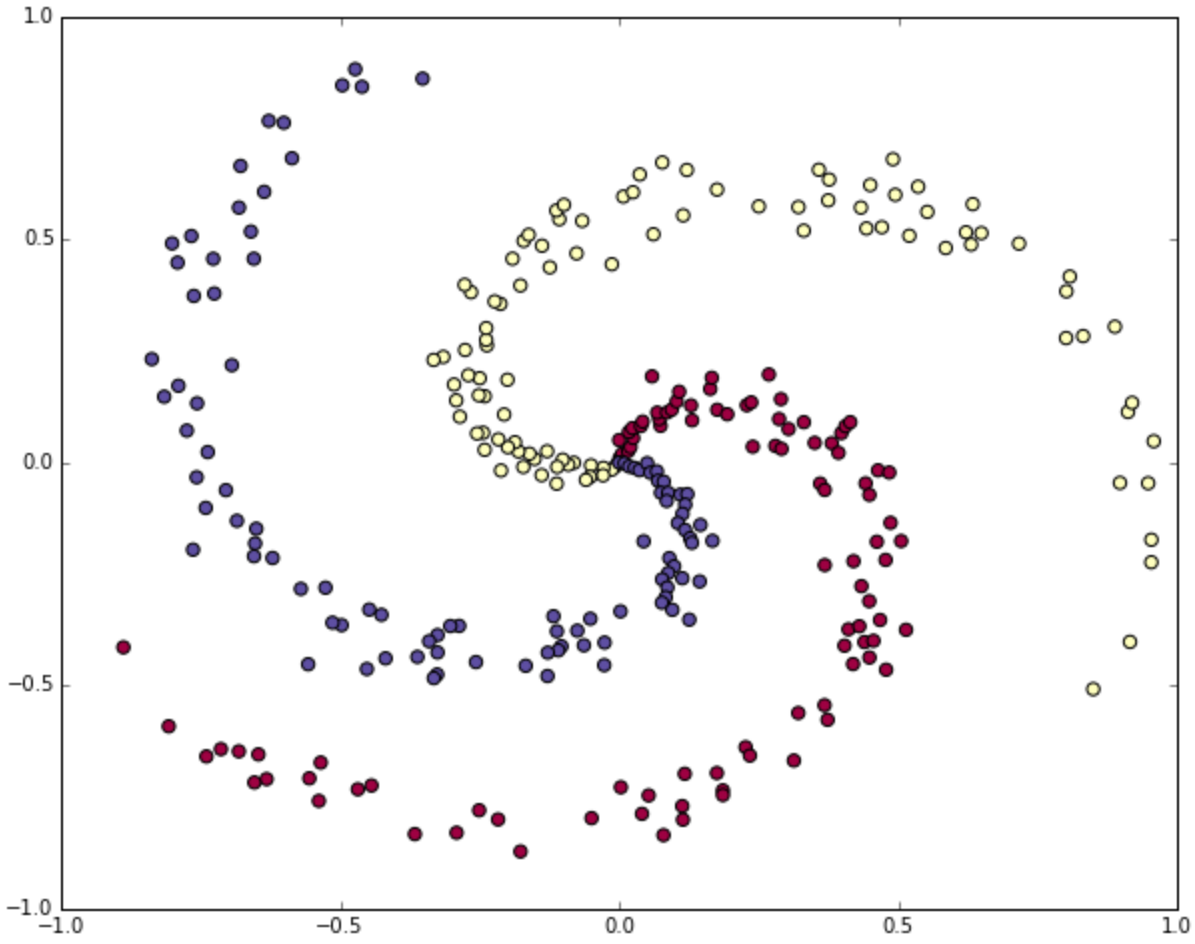
In this section we'll walk through a complete implementation of a toy Neural Network in 2 dimensions. We'll first implement a simple linear classifier and then extend the code to a 2-layer Neural Network. As we'll see, this extension is surprisingly simple and very few changes are necessary.

Generating some data

Lets generate a classification dataset that is not easily linearly separable. Our favorite example is the spiral dataset, which can be generated as follows:

```
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels
for j in range(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
  y[ix] = j
# lets visualize the data:
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.show()
```



The toy spiral data consists of three classes (blue, red, yellow) that are not linearly separable.

Normally we would want to preprocess the dataset so that each feature has zero mean and unit standard deviation, but in this case the features are already in a nice range from -1 to 1, so we skip this step.

Training a Softmax Linear Classifier

Initialize the parameters

Lets first train a Softmax classifier on this classification dataset. As we saw in the previous sections, the Softmax classifier has a linear score function and uses the cross-entropy loss. The parameters of the linear classifier consist of a weight matrix `W` and a bias vector `b` for each class. Lets first initialize these parameters to be random numbers:

```
# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))
```

Recall that we `D = 2` is the dimensionality and `K = 3` is the number of classes.

Compute the class scores

Since this is a linear classifier, we can compute all class scores very simply in parallel with a single matrix multiplication:

```
# compute class scores for a linear classifier
scores = np.dot(X, W) + b
```

In this example we have 300 2-D points, so after this multiplication the array `scores` will have size [300 x 3], where each row gives the class scores corresponding to the 3 classes (blue, red, yellow).

Compute the loss

The second key ingredient we need is a loss function, which is a differentiable objective that quantifies our unhappiness with the computed class scores. Intuitively, we want the correct class to have a higher score than the other classes. When this is the case, the loss should be low and otherwise the loss should be high. There are many ways to quantify this intuition, but in this example lets use the cross-entropy loss that is associated with the Softmax classifier. Recall that if f is the array of class scores for a single example (e.g. array of 3 numbers here), then the Softmax classifier computes the loss for that example as:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

We can see that the Softmax classifier interprets every element of f as holding the (unnormalized) log probabilities of the three classes. We exponentiate these to get (unnormalized) probabilities, and then normalize them to get probabilites. Therefore, the expression inside the log is the normalized probability of the correct class. Note how this expression works: this quantity is always between 0 and 1. When the probability of the correct class is very small (near 0), the loss will go towards (positive) infinity. Conversely, when the correct class probability goes towards 1, the loss will go towards zero because $log(1) = 0$. Hence, the expression for L_i is low when the correct class probability is high, and it's very high when it is low.

Recall also that the full Softmax classifier loss is then defined as the average cross-entropy loss over the training examples and the regularization:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\frac{1}{2}\lambda\sum_k\sum_l W_{k,l}^2}_{\text{regularization loss}}$$

Given the array of `scores` we've computed above, we can compute the loss. First, the way to obtain the probabilities is straight forward:

```
num_examples = X.shape[0]
# get unnormalized probabilities
exp_scores = np.exp(scores)
# normalize them for each example
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

We now have an array `probs` of size [300 x 3], where each row now contains the class probabilities. In particular, since we've normalized them every row now sums to one. We can now query for the log probabilities assigned to the correct classes in each example:

```
correct_logprobs = -np.log(probs[range(num_examples),y])
```

The array `correct_logprobs` is a 1D array of just the probabilities assigned to the correct classes for each example. The full loss is then the average of these log probabilities and the regularization loss:

```
# compute the loss: average cross-entropy loss and regularization
data_loss = np.sum(correct_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
```

In this code, the regularization strength λ is stored inside the `reg`. The convenience factor of `0.5` multiplying the regularization will become clear in a second. Evaluating this in the beginning (with random parameters) might give us `loss = 1.1`, which is `-np.log(1.0/3)`, since with small initial random weights all probabilities assigned to all classes are about one third. We now want to make the loss as low as possible, with `loss = 0` as the absolute lower bound. But the lower the loss is, the higher are the probabilities assigned to the correct classes for all examples.

Computing the Analytic Gradient with Backpropagation

We have a way of evaluating the loss, and now we have to minimize it. We'll do so with gradient descent. That is, we start with random parameters (as shown above), and evaluate the gradient of the loss function with respect to the parameters, so that we know how we should change the parameters to decrease the loss. Lets introduce the intermediate variable p, which is a vector of the (normalized) probabilities. The loss for one example is:

$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \qquad\qquad L_i = -\log\left(p_{y_i}\right)$$

We now wish to understand how the computed scores inside f should change to decrease the loss L_i that this example contributes to the full objective. In other words, we want to derive the gradient $\partial L_i / \partial f_k$. The loss L_i is computed from p, which in turn depends on f. It's a fun exercise to the reader to use the chain rule to derive the gradient, but it turns out to be extremely simple and interpretible in the end, after a lot of things cancel out:

$$\frac{\partial L_i}{\partial f_k} = p_k - 1(y_i = k)$$

Notice how elegant and simple this expression is. Suppose the probabilities we computed were `p = [0.2, 0.3, 0.5]`, and that the correct class was the middle one (with probability 0.3). According to this derivation the gradient on the scores would be `df = [0.2, -0.7, 0.5]`. Recalling what the interpretation of the gradient, we see that this result is highly intuitive: increasing the first or last element of the score vector `f` (the scores of the incorrect classes) leads to an *increased* loss (due to the positive signs +0.2 and +0.5) - and increasing the loss is bad, as expected. However, increasing the score of the correct class has *negative* influence on the loss. The gradient of -0.7 is telling us that increasing the correct class score would lead to a decrease of the loss L_i, which makes sense.

All of this boils down to the following code. Recall that `probs` stores the probabilities of all classes (as rows) for each example. To get the gradient on the scores, which we call `dscores`, we proceed as follows:

```
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples
```

Lastly, we had that `scores = np.dot(X, W) + b`, so armed with the gradient on `scores` (stored in `dscores`), we can now backpropagate into `W` and `b`:

```
dW = np.dot(X.T, dscores)
db = np.sum(dscores, axis=0, keepdims=True)
dW += reg*W # don't forget the regularization gradient
```

Where we see that we have backpropped through the matrix multiply operation, and also added the contribution from the regularization. Note that the regularization gradient has the very simple form `reg*W` since we used the constant `0.5` for its loss contribution (i.e. $\frac{d}{dw}(\frac{1}{2}\lambda w^2) = \lambda w$. This is a common convenience trick that simplifies the gradient expression.

Performing a parameter update

Now that we've evaluated the gradient we know how every parameter influences the loss function. We will now perform a parameter update in the *negative* gradient direction to *decrease* the loss:

```
# perform a parameter update
W += -step_size * dW
b += -step_size * db
```

Putting it all together: Training a Softmax Classifier

Putting all of this together, here is the full code for training a Softmax classifier with Gradient descent:

```
#Train a Linear Classifier

# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in range(200):

  # evaluate class scores, [N x K]
  scores = np.dot(X, W) + b

  # compute the class probabilities
  exp_scores = np.exp(scores)
  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

  # compute the loss: average cross-entropy loss and regularization
  correct_logprobs = -np.log(probs[range(num_examples),y])
  data_loss = np.sum(correct_logprobs)/num_examples
  reg_loss = 0.5*reg*np.sum(W*W)
  loss = data_loss + reg_loss
  if i % 10 == 0:
    print "iteration %d: loss %f" % (i, loss)

  # compute the gradient on scores
  dscores = probs
  dscores[range(num_examples),y] -= 1
  dscores /= num_examples
```

```
  # backpropate the gradient to the parameters (W,b)
  dW = np.dot(X.T, dscores)
  db = np.sum(dscores, axis=0, keepdims=True)

  dW += reg*W # regularization gradient

  # perform a parameter update
  W += -step_size * dW
  b += -step_size * db
```

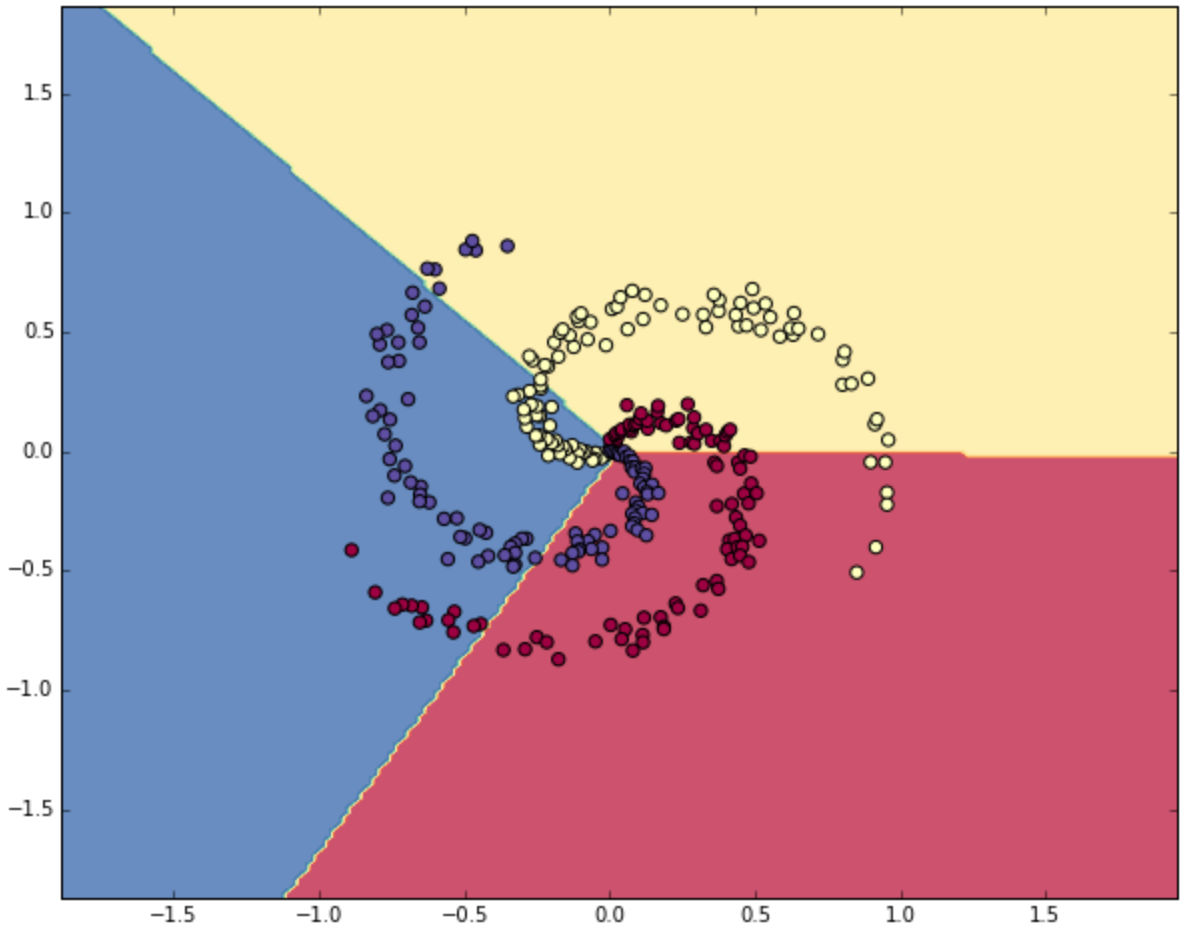
Running this prints the output:

```
iteration 0: loss 1.096956
iteration 10: loss 0.917265
iteration 20: loss 0.851503
iteration 30: loss 0.822336
iteration 40: loss 0.807586
iteration 50: loss 0.799448
iteration 60: loss 0.794681
iteration 70: loss 0.791764
iteration 80: loss 0.789920
iteration 90: loss 0.788726
iteration 100: loss 0.787938
iteration 110: loss 0.787409
iteration 120: loss 0.787049
iteration 130: loss 0.786803
iteration 140: loss 0.786633
iteration 150: loss 0.786514
iteration 160: loss 0.786431
iteration 170: loss 0.786373
iteration 180: loss 0.786331
iteration 190: loss 0.786302
```

We see that we've converged to something after about 190 iterations. We can evaluate the training set accuracy:

```
# evaluate training set accuracy
scores = np.dot(X, W) + b
predicted_class = np.argmax(scores, axis=1)
print 'training accuracy: %.2f' % (np.mean(predicted_class == y))
```

This prints **49%**. Not very good at all, but also not surprising given that the dataset is constructed so it is not linearly separable. We can also plot the learned decision boundaries:



Linear classifier fails to learn the toy spiral dataset.

Training a Neural Network

Clearly, a linear classifier is inadequate for this dataset and we would like to use a Neural Network. One additional hidden layer will suffice for this toy data. We will now need two sets of weights and biases (for the first and second layers):

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))
```

The forward pass to compute scores now changes form:

```
# evaluate class scores with a 2-layer Neural Network
hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
scores = np.dot(hidden_layer, W2) + b2
```

Notice that the only change from before is one extra line of code, where we first compute the hidden layer representation and then the scores based on this hidden layer. Crucially, we've also added a non-linearity, which in this case is simple ReLU that thresholds the activations on the hidden layer at zero.

Everything else remains the same. We compute the loss based on the scores exactly as before, and get the gradient for the scores `dscores` exactly as before. However, the way we backpropagate that gradient into the model parameters now changes form, of course. First lets backpropagate the second layer of the Neural Network. This looks identical to the code we had for the Softmax classifier, except we're replacing `X` (the raw data), with the variable `hidden_layer`):

```
# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, dscores)
db2 = np.sum(dscores, axis=0, keepdims=True)
```

However, unlike before we are not yet done, because `hidden_layer` is itself a function of other parameters and the data! We need to continue backpropagation through this variable. Its gradient can be computed as:

```
dhidden = np.dot(dscores, W2.T)
```

Now we have the gradient on the outputs of the hidden layer. Next, we have to backpropagate the ReLU non-linearity. This turns out to be easy because ReLU during the backward pass is effectively a switch. Since $r = max(0, x)$, we have that $\frac{dr}{dx} = 1(x > 0)$. Combined with the chain rule, we see that the ReLU unit lets the gradient pass through unchanged if its input was greater than 0, but *kills it* if its input was less than zero during the forward pass. Hence, we can backpropagate the ReLU in place simply with:

```
# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0
```

And now we finally continue to the first layer weights and biases:

```
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)
```

We're done! We have the gradients `dW,db,dW2,db2` and can perform the parameter update. Everything changed. The full code looks very similar:

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in range(10000):

  # evaluate class scores, [N x K]
  hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
  scores = np.dot(hidden_layer, W2) + b2

  # compute the class probabilities
  exp_scores = np.exp(scores)
  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

  # compute the loss: average cross-entropy loss and regularization
  correct_logprobs = -np.log(probs[range(num_examples),y])
  data_loss = np.sum(correct_logprobs)/num_examples
  reg_loss = 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2)
  loss = data_loss + reg_loss
  if i % 1000 == 0:
    print "iteration %d: loss %f" % (i, loss)

  # compute the gradient on scores
  dscores = probs
  dscores[range(num_examples),y] -= 1
  dscores /= num_examples

  # backpropate the gradient to the parameters
  # first backprop into parameters W2 and b2
  dW2 = np.dot(hidden_layer.T, dscores)
  db2 = np.sum(dscores, axis=0, keepdims=True)
  # next backprop into hidden layer
  dhidden = np.dot(dscores, W2.T)
  # backprop the ReLU non-linearity
  dhidden[hidden_layer <= 0] = 0
  # finally into W,b
  dW = np.dot(X.T, dhidden)
```

```
  db = np.sum(dhidden, axis=0, keepdims=True)

  # add regularization gradient contribution
  dW2 += reg * W2
  dW += reg * W

  # perform a parameter update
  W += -step_size * dW
  b += -step_size * db
  W2 += -step_size * dW2
  b2 += -step_size * db2
```

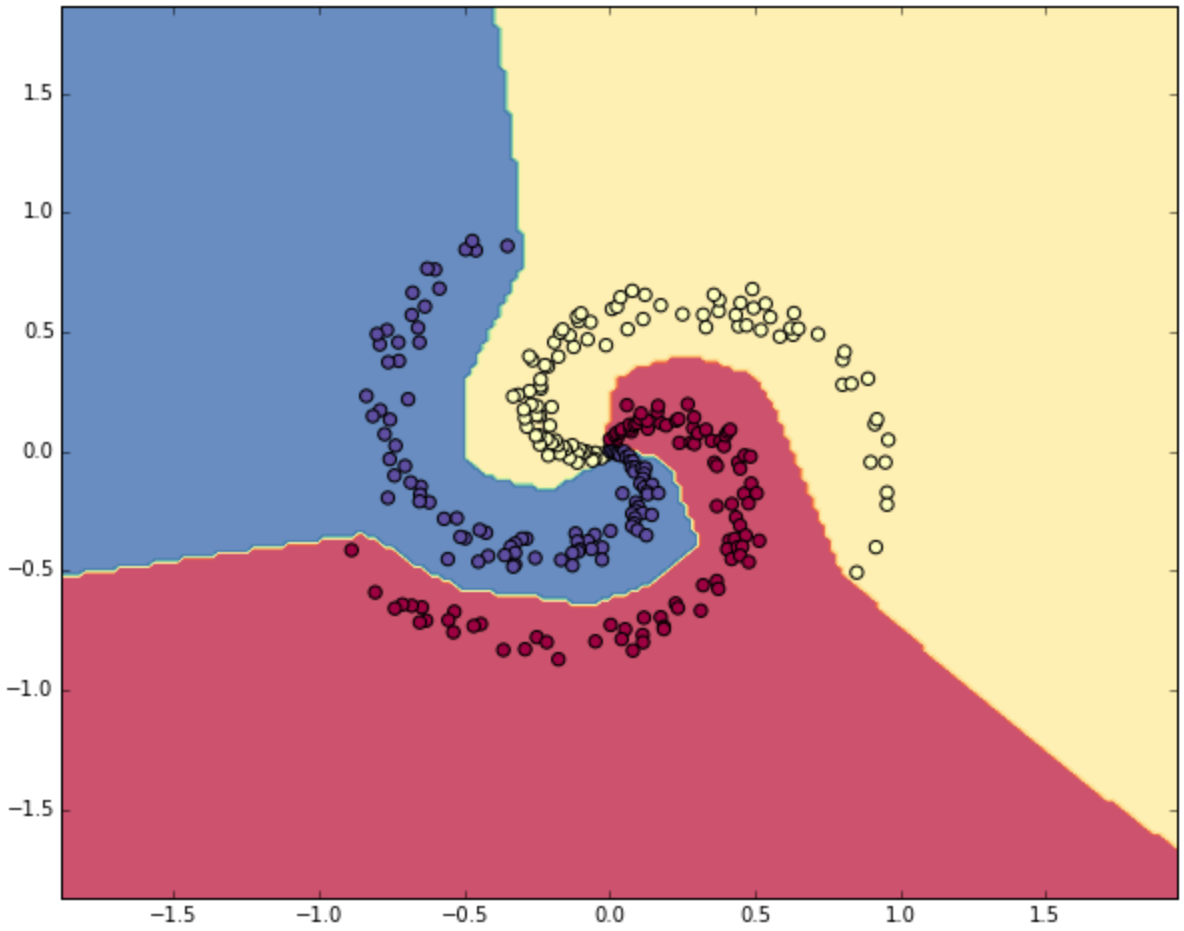
This prints:

```
iteration 0: loss 1.098744
iteration 1000: loss 0.294946
iteration 2000: loss 0.259301
iteration 3000: loss 0.248310
iteration 4000: loss 0.246170
iteration 5000: loss 0.245649
iteration 6000: loss 0.245491
iteration 7000: loss 0.245400
iteration 8000: loss 0.245335
iteration 9000: loss 0.245292
```

The training accuracy is now:

```
# evaluate training set accuracy
hidden_layer = np.maximum(0, np.dot(X, W) + b)
scores = np.dot(hidden_layer, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print 'training accuracy: %.2f' % (np.mean(predicted_class == y))
```

Which prints **98%**!. We can also visualize the decision boundaries:



Neural Network classifier crushes the spiral dataset.

Summary

We've worked with a toy 2D dataset and trained both a linear network and a 2-layer Neural Network. We saw that the change from a linear classifier to a Neural Network involves very few changes in the code. The score function changes its form (1 line of code difference), and the backpropagation changes its form (we have to perform one more round of backprop through the hidden layer to the first layer of the network).

- You may want to look at this IPython Notebook code rendered as HTML.
- Or download the ipynb file

Table of Contents:

Convolutional Neural Networks (CNNs / ConvNets)

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. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

So what changes? 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.

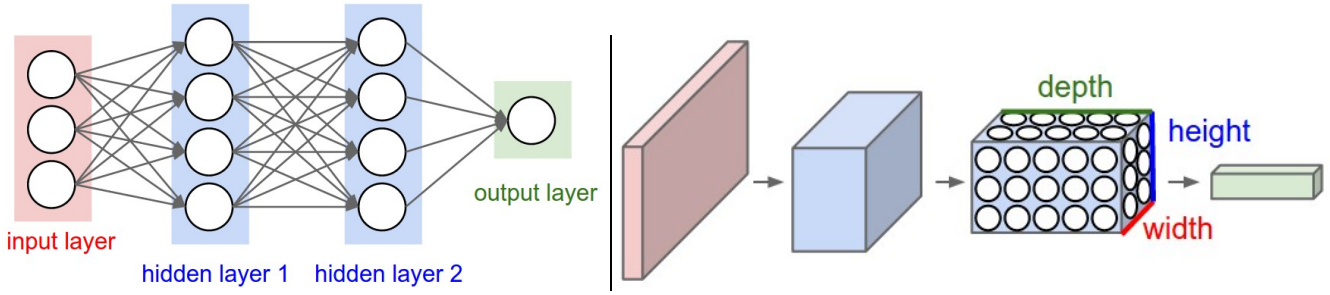
Architecture Overview

Recall: Regular Neural Nets. As we saw in the previous chapter, Neural Networks receive an input (a single vector), and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer

function completely independently and do not share any connections. The last fully-connected layer is called the "output layer" and in classification settings it represents the class scores.

Regular Neural Nets don't scale well to full images. In CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

3D volumes of neurons. 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). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:



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).

> *A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters.*

Layers used to build ConvNets

As we described above, a simple ConvNet is a sequence of layers, and 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**.

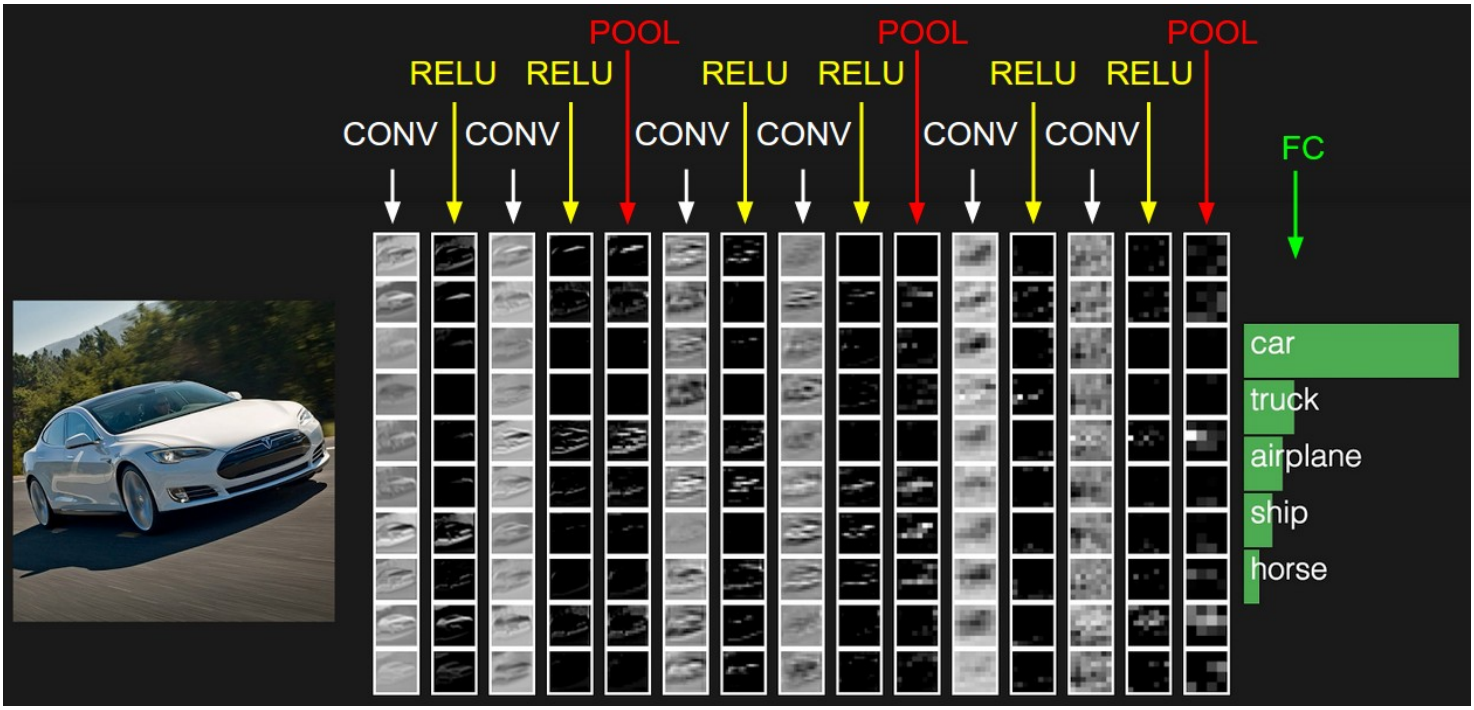
Example Architecture: Overview. We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the $max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

In summary:

- A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)



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.

We now describe the individual layers and the details of their hyperparameters and their connectivities.

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. Let's first discuss what the CONV layer computes without brain/neuron analogies. 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 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.

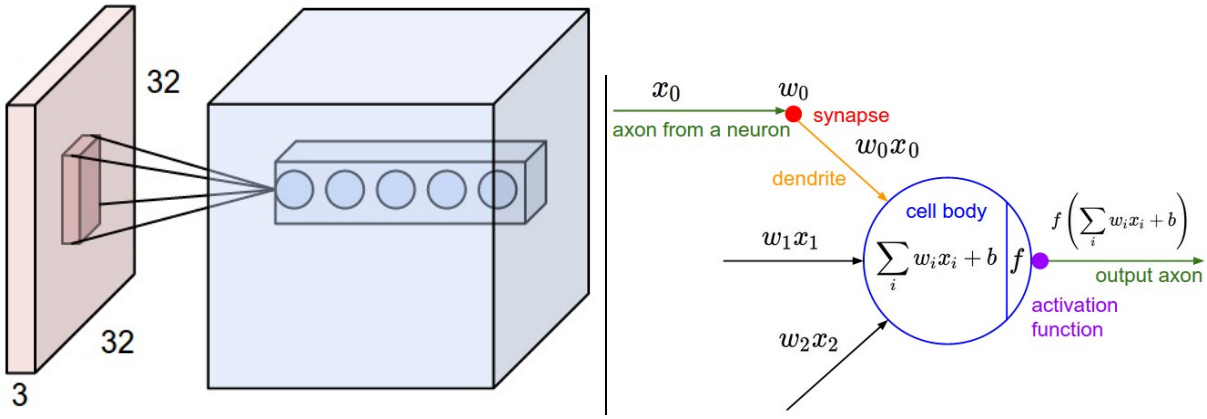
The brain view. If you're a fan of the brain/neuron analogies, every entry in the 3D output volume can also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (since these numbers all result from applying the same filter).

We now discuss the details of the neuron connectivities, their arrangement in space, and their parameter sharing scheme.

Local Connectivity. When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in 2D space (along width and height), but always full along the entire depth of the input volume.

Example 1. For example, suppose that the input volume has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights (and +1 bias parameter). Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.

Example 2. Suppose an input volume had size [16x16x20]. Then using an example receptive field size of 3x3, every neuron in the Conv Layer would now have a total of 3*3*20 = 180 connections to the input volume. Notice that, again, the connectivity is local in 2D space (e.g. 3x3), but full along the input depth (20).



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: the lines that connect this column of 5 neurons do not represent the weights (i.e. these 5 neurons do not share the same weights, but they are associated with 5 different filters), they just indicate that these neurons are connected to or looking at the same receptive field or region of the input volume, i.e. they share the same receptive field but not the same weights. **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.

Spatial arrangement. We have explained the connectivity of each neuron in the Conv Layer to the input volume, but we haven't yet discussed how many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume: the **depth, stride** and **zero-padding**. We discuss these next:

1. 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. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column** (some people also prefer the term *fibre*).
2. 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.
3. As we will soon see, 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 we'll 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).

We can compute the spatial size of the output volume as a function of the input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. You can convince yourself that the correct formula for calculating how many neurons "fit" is given by $(W - F + 2P)/S + 1$. For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output. Lets also see one more graphical example:

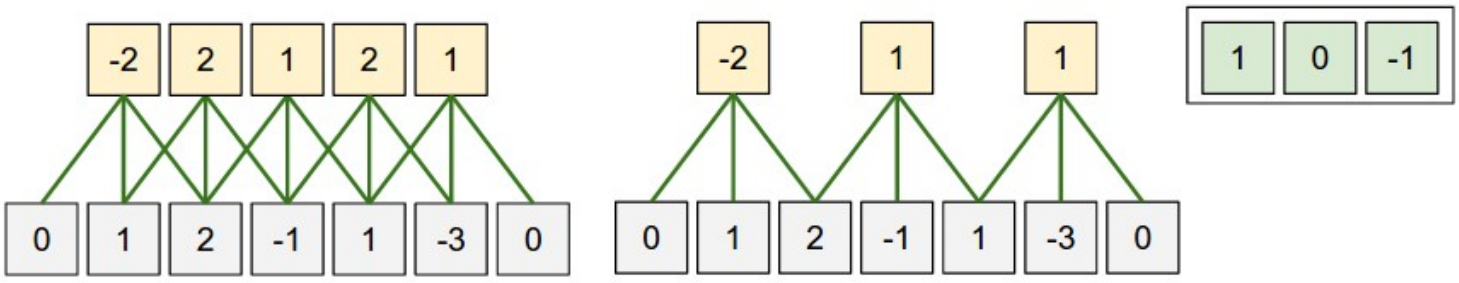


Illustration of spatial arrangement. In this example there is only one spatial dimension (x-axis), one neuron with a receptive field size of F = 3, the input size is W = 5, and there is zero padding of P = 1. **Left:** The neuron strided across the input in stride of S = 1, giving output of size (5 - 3 + 2)/1+1 = 5. **Right:** The neuron uses stride of S = 2, giving output of size (5 - 3 + 2)/2+1 = 3. Notice that stride S = 3 could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since (5 - 3 + 2) = 4 is not divisible by 3.
The neuron weights are in this example [1,0,-1] (shown on very right), and its bias is zero. These weights are shared across all yellow neurons (see parameter sharing below).

Use of zero-padding. In the example above on left, note that the input dimension was 5 and the output dimension was equal: also 5. This worked out so because our receptive fields were 3 and we used zero padding of 1. If there was no zero-padding used, then the output volume would have had spatial dimension of only 3, because that is how many neurons would have "fit" across the original input. In general, setting zero padding to be $P = (F - 1)/2$ when the stride is $S = 1$ ensures that the input volume and output volume will have the same size spatially. It is very common to use zero-padding in this way and we will discuss the full reasons when we talk more about ConvNet architectures.

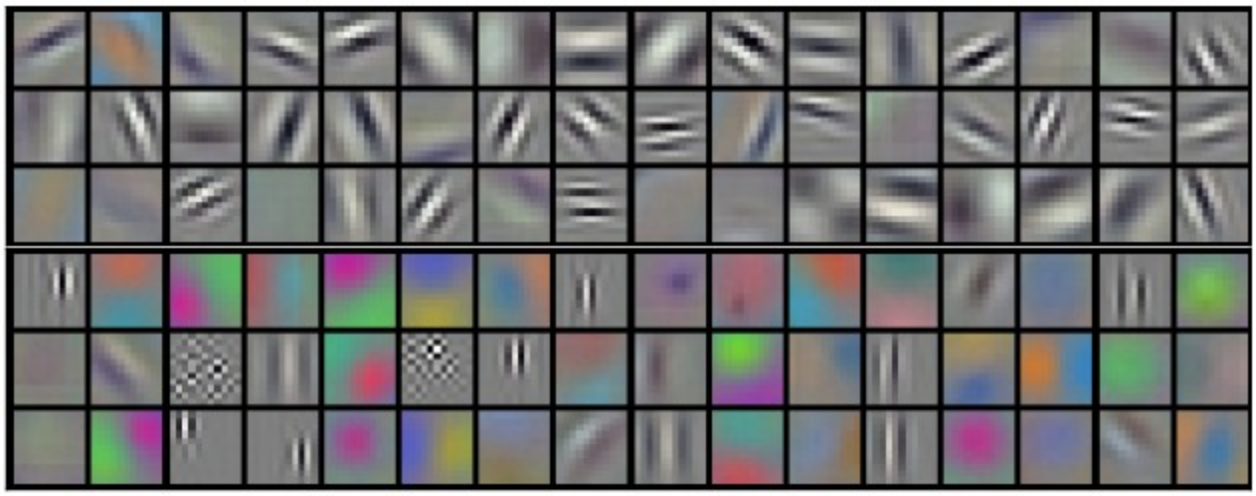
Constraints on strides. Note again that the spatial arrangement hyperparameters have mutual constraints. For example, when the input has size $W = 10$, no zero-padding is used $P = 0$, and the filter size is $F = 3$, then it would be impossible to use stride $S = 2$, since $(W - F + 2P)/S + 1 = (10 - 3 + 0)/2 + 1 = 4.5$, i.e. not an integer, indicating that the neurons don't "fit" neatly and symmetrically across the input. Therefore, this setting of the hyperparameters is considered to be invalid, and a ConvNet library could throw an exception or zero pad the rest to make it fit, or crop the input to make it fit, or something. As we will see in the ConvNet architectures section, sizing the ConvNets appropriately so that all the dimensions "work out" can be a real headache, which the use of zero-padding and some design guidelines will significantly alleviate.

Real-world example. The Krizhevsky et al. architecture that won the ImageNet challenge in 2012 accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field size $F = 11$, stride $S = 4$ and no zero padding $P = 0$. Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of $K = 96$, the Conv layer output volume had size [55x55x96]. Each of the 55*55*96 neurons in this volume was connected to a region of size [11x11x3] in the input volume. Moreover, all 96 neurons in each depth column are connected to the same [11x11x3] region of the input, but of course with different weights. As a fun aside, if you read the actual paper it claims that the input images were 224x224, which is surely incorrect because (224 - 11)/4 + 1 is quite clearly not an integer. This has confused many people in the history of ConvNets and little is known about what happened. My own best guess is that Alex used zero-padding of 3 extra pixels that he does not mention in the paper.

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). Alternatively, all 55*55 neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the CONV layer can in each depth slice be computed as a **convolution** of the neuron's weights with the input volume (Hence the name: Convolutional Layer). This is why it is common to refer to the sets of weights as a **filter** (or a **kernel**), that is convolved with the input.



Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

Note that sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a ConvNet have some specific centered structure, where we should expect, for example, that completely different features should be learned on one side of the image than another. One practical example is when the input are faces that have been centered in the image. You might expect that different eye-specific or hair-specific features could (and should) be learned in different spatial locations. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a **Locally-Connected Layer**.

Numpy examples. To make the discussion above more concrete, lets express the same ideas but in code and with a specific example. Suppose that the input volume is a numpy array `X`. Then:

- A *depth column* (or a *fibre*) at position `(x,y)` would be the activations `X[x,y,:]`.
- A *depth slice*, or equivalently an *activation map* at depth `d` would be the activations `X[:,:,d]`.

Conv Layer Example. Suppose that the input volume `X` has shape `X.shape: (11,11,4)`. Suppose further that we use no zero padding ($P = 0$), that the filter size is $F = 5$, and that the stride is $S = 2$. The output volume would therefore have spatial size (11-5)/2+1 = 4, giving a volume with width and height of 4. The activation map in the output volume (call it `V`), would then look as follows (only some of the elements are computed in this example):

- `V[0,0,0] = np.sum(X[:5,:5,:] * W0) + b0`
- `= np.sum(X[2:7,:5,:] * W0) + b0`

- `V[2,0,0] = np.sum(X[4:9,:5,:] * W0) + b0`
- `V[3,0,0] = np.sum(X[6:11,:5,:] * W0) + b0`

Remember that in numpy, the operation `*` above denotes elementwise multiplication between the arrays. Notice also that the weight vector `W0` is the weight vector of that neuron and `b0` is the bias. Here, `W0` is assumed to be of shape `W0.shape: (5,5,4)`, since the filter size is 5 and the depth of the input volume is 4. Notice that at each point, we are computing the dot product as seen before in ordinary neural networks. Also, we see that we are using the same weight and bias (due to parameter sharing), and where the dimensions along the width are increasing in steps of 2 (i.e. the stride). To construct a second activation map in the output volume, we would have:

- `V[0,0,1] = np.sum(X[:5,:5,:] * W1) + b1`
- `V[1,0,1] = np.sum(X[2:7,:5,:] * W1) + b1`
- `V[2,0,1] = np.sum(X[4:9,:5,:] * W1) + b1`
- `V[3,0,1] = np.sum(X[6:11,:5,:] * W1) + b1`
- `V[0,1,1] = np.sum(X[:5,2:7,:] * W1) + b1` (example of going along y)
- `V[2,3,1] = np.sum(X[4:9,6:11,:] * W1) + b1` (or along both)

where we see that we are indexing into the second depth dimension in `V` (at index 1) because we are computing the second activation map, and that a different set of parameters (`W1`) is now used. In the example above, we are for brevity leaving out some of the other operations the Conv Layer would perform to fill the other parts of the output array `V`. Additionally, recall that these activation maps are often followed elementwise through an activation function such as ReLU, but this is not shown here.

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

A common setting of the hyperparameters is $F = 3, S = 1, P = 1$. However, there are common conventions and rules of thumb that motivate these hyperparameters. See the ConvNet architectures section below.

Convolution Demo. Below is a running demo of a CONV layer. Since 3D volumes are hard to visualize, all the volumes (the input volume (in blue), the weight volumes (in red), the output volume (in green)) are visualized with each depth slice stacked in rows. The input volume is of size $W_1 = 5, H_1 = 5, D_1 = 3$, and the CONV layer parameters are $K = 2, F = 3, S = 2, P = 1$. That is, we have two filters of size 3×3, and they are applied with a stride of 2. Therefore, the output volume size has spatial size (5 - 3 + 2)/2 + 1 = 3. Moreover, notice that a padding of $P = 1$ is applied to the input volume, making the outer border of the input volume zero. The visualization below iterates over the output activations (green), and shows that each element is computed by elementwise multiplying the highlighted input (blue) with the filter (red), summing it up, and then offsetting the result by the bias.

Input Volume (+pad 1) (7x7x3)

`x[:,:,0]`

0	0	0	0	0	0	0
0	1	0	0	0	2	0
0	2	2	1	2	2	0
0	2	2	2	0	0	0
0	1	2	1	1	0	0
0	2	0	1	1	2	0
0	0	0	0	0	0	0

`x[:,:,1]`

0	0	0	0	0	0	0
0	1	1	2	1	0	0
0	2	1	1	2	1	0
0	0	2	2	0	1	0
0	1	2	2	1	2	0
0	2	2	2	1	0	0
0	0	0	0	0	0	0

`x[:,:,2]`

0	0	0	0	0	0	0
0	2	2	2	1	2	0
0	1	0	2	1	0	0
0	2	1	1	1	1	0
0	0	1	2	2	2	0
0	0	1	0	1	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

`w0[:,:,0]`

-1	0	-1
1	0	-1
1	-1	0

`w0[:,:,1]`

-1	0	1
1	1	-1
0	0	-1

`w0[:,:,2]`

-1	1	1
1	0	0
-1	0	0

Bias b0 (1x1x1)

`b0[:,:,0]`

1

Filter W1 (3x3x3)

`w1[:,:,0]`

0	-1	1
0	-1	-1
1	-1	-1

`w1[:,:,1]`

-1	0	-1
1	-1	0
-1	1	1

`w1[:,:,2]`

1	-1	-1
1	-1	1
1	-1	0

Bias b1 (1x1x1)

`b1[:,:,0]`

0

Output Volume (3x3x2)

`o[:,:,0]`

-2	4	2
-6	7	-3
2	3	2

`o[:,:,1]`

-4	-1	-2
-7	-6	-2
-5	-6	-3

toggle movement

Implementation as Matrix Multiplication. Note that the convolution operation essentially performs dot

n the filters and local regions of the input. A common implementation pattern of the CONV

layer is to take advantage of this fact and formulate the forward pass of a convolutional layer as one big matrix multiply as follows:

1. The local regions in the input image are stretched out into columns in an operation commonly called **im2col**. For example, if the input is [227x227x3] and it is to be convolved with 11x11x3 filters at stride 4, then we would take [11x11x3] blocks of pixels in the input and stretch each block into a column vector of size 11*11*3 = 363. Iterating this process in the input at stride of 4 gives (227-11)/4+1 = 55 locations along both width and height, leading to an output matrix `X_col` of *im2col* of size [363 x 3025], where every column is a stretched out receptive field and there are 55*55 = 3025 of them in total. Note that since the receptive fields overlap, every number in the input volume may be duplicated in multiple distinct columns.
2. The weights of the CONV layer are similarly stretched out into rows. For example, if there are 96 filters of size [11x11x3] this would give a matrix `W_row` of size [96 x 363].
3. The result of a convolution is now equivalent to performing one large matrix multiply `np.dot(W_row, X_col)`, which evaluates the dot product between every filter and every receptive field location. In our example, the output of this operation would be [96 x 3025], giving the output of the dot product of each filter at each location.
4. The result must finally be reshaped back to its proper output dimension [55x55x96].

This approach has the downside that it can use a lot of memory, since some values in the input volume are replicated multiple times in `X_col`. However, the benefit is that there are many very efficient implementations of Matrix Multiplication that we can take advantage of (for example, in the commonly used BLAS API). Moreover, the same *im2col* idea can be reused to perform the pooling operation, which we discuss next.

Backpropagation. The backward pass for a convolution operation (for both the data and the weights) is also a convolution (but with spatially-flipped filters). This is easy to derive in the 1-dimensional case with a toy example (not expanded on for now).

1x1 convolution. As an aside, several papers use 1x1 convolutions, as first investigated by Network in Network. Some people are at first confused to see 1x1 convolutions especially when they come from signal processing background. Normally signals are 2-dimensional so 1x1 convolutions do not make sense (it's just pointwise scaling). However, in ConvNets this is not the case because one must remember that we operate over 3-dimensional volumes, and that the filters always extend through the full depth of the input volume. For example, if the input is [32x32x3] then doing 1x1 convolutions would effectively be doing 3-dimensional dot products (since the input depth is 3 channels).

Dilated convolutions. A recent development (e.g. see paper by Fisher Yu and Vladlen Koltun) is to introduce one more hyperparameter to the CONV layer called the *dilation*. So far we've only discussed CONV filters that are contiguous. However, it's possible to have filters that have spaces between each cell, called dilation. As an example, in one dimension a filter `w` of size 3 would compute over input `x` the following: `w[0]*x[0] + w[1]*x[1] + w[2]*x[2]`. This is dilation of 0. For dilation 1 the filter would instead compute `w[0]*x[0] + w[1]*x[2] + w[2]*x[4]`; In other words there is a gap of 1 between the applications. This can be very useful in some settings to use in conjunction with 0-dilated filters because it allows you to

merge spatial information across the inputs much more agressively with fewer layers. For example, if you stack two 3x3 CONV layers on top of each other then you can convince yourself that the neurons on the 2nd layer are a function of a 5x5 patch of the input (we would say that the *effective receptive field* of these neurons is 5x5). If we use dilated convolutions then this effective receptive field would grow much quicker.

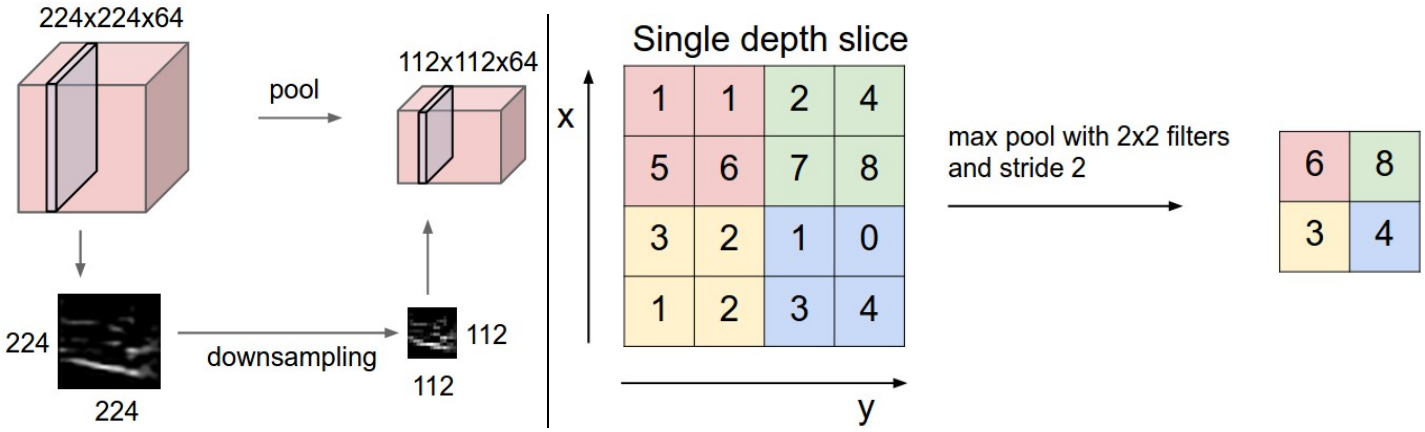
Pooling Layer

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 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. More generally, the pooling layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- 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.

It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: A pooling layer with $F = 3, S = 2$ (also called overlapping pooling), and more commonly $F = 2, S = 2$. Pooling sizes with larger receptive fields are too destructive.

General pooling. 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.



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).

Backpropagation. Recall from the backpropagation chapter that the backward pass for a max(x, y) operation has a simple interpretation as only routing the gradient to the input that had the highest value in the forward pass. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called *the switches*) so that gradient routing is efficient during backpropagation.

Getting rid of pooling. Many people dislike the pooling operation and think that we can get away without it. For example, Striving for Simplicity: The All Convolutional Net proposes to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs). It seems likely that future architectures will feature very few to no pooling layers.

Normalization Layer

Many types of normalization layers have been proposed for use in ConvNet architectures, sometimes with the intentions of implementing inhibition schemes observed in the biological brain. However, these layers have since fallen out of favor because in practice their contribution has been shown to be minimal, if any. For various types of normalizations, see the discussion in Alex Krizhevsky's cuda-convnet library API.

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.

Converting FC layers to CONV layers

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. Therefore, it turns out that it's possible to convert between FC and CONV layers:

- For any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing).
- Conversely, any FC layer can be converted to a CONV layer. For example, an FC layer with $K = 4096$ that is looking at some input volume of size $7 \times 7 \times 512$ can be equivalently expressed as a CONV layer with $F = 7, P = 0, S = 1, K = 4096$. In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be $1 \times 1 \times 4096$ since only a single depth column "fits" across the input volume, giving identical result as the initial FC layer.

FC->CONV conversion. Of these two conversions, the ability to convert an FC layer to a CONV layer is particularly useful in practice. Consider a ConvNet architecture that takes a 224x224x3 image, and then uses a series of CONV layers and POOL layers to reduce the image to an activations volume of size 7x7x512 (in an *AlexNet* architecture that we'll see later, this is done by use of 5 pooling layers that downsample the input spatially by a factor of two each time, making the final spatial size 224/2/2/2/2/2 = 7). From there, an AlexNet uses two FC layers of size 4096 and finally the last FC layers with 1000 neurons that compute the class scores. We can convert each of these three FC layers to CONV layers as described above:

- Replace the first FC layer that looks at [7x7x512] volume with a CONV layer that uses filter size $F = 7$, giving output volume [1x1x4096].
- Replace the second FC layer with a CONV layer that uses filter size $F = 1$, giving output volume [1x1x4096]
- Replace the last FC layer similarly, with $F = 1$, giving final output [1x1x1000]

Each of these conversions could in practice involve manipulating (e.g. reshaping) the weight matrix W in each FC layer into CONV layer filters. It turns out that this conversion allows us to "slide" the original ConvNet very efficiently across many spatial positions in a larger image, in a single forward pass.

For example, if 224x224 image gives a volume of size [7x7x512] - i.e. a reduction by 32, then forwarding an image of size 384x384 through the converted architecture would give the equivalent volume in size [12x12x512], since 384/32 = 12. Following through with the next 3 CONV layers that we just converted from FC layers would now give the final volume of size [6x6x1000], since (12 - 7)/1 + 1 = 6. Note that instead of a single vector of class scores of size [1x1x1000], we're now getting an entire 6x6 array of class scores across the 384x384 image.

> *Evaluating the original ConvNet (with FC layers) independently across 224x224 crops of the 384x384 image in strides of 32 pixels gives an identical result to forwarding the converted ConvNet one time.*

Naturally, forwarding the converted ConvNet a single time is much more efficient than iterating the original those 36 locations, since the 36 evaluations share computation. This trick is often used in

practice to get better performance, where for example, it is common to resize an image to make it bigger, use a converted ConvNet to evaluate the class scores at many spatial positions and then average the class scores.

Lastly, what if we wanted to efficiently apply the original ConvNet over the image but at a stride smaller than 32 pixels? We could achieve this with multiple forward passes. For example, note that if we wanted to use a stride of 16 pixels we could do so by combining the volumes received by forwarding the converted ConvNet twice: First over the original image and second over the image but with the image shifted spatially by 16 pixels along both width and height.

- An IPython Notebook on Net Surgery shows how to perform the conversion in practice, in code (using Caffe)

ConvNet Architectures

We have seen that Convolutional Networks are commonly made up of only three layer types: CONV, POOL (we assume Max pool unless stated otherwise) and FC (short for fully-connected). We will also explicitly write the RELU activation function as a layer, which applies elementwise non-linearity. In this section we discuss how these are commonly stacked together to form entire ConvNets.

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 >= 0` (and usually `N <= 3`), `M >= 0`, `K >= 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.

Prefer a stack of small filter CONV to one large receptive field CONV layer. Suppose that you stack three 3x3 CONV layers on top of each other (with non-linearities in between, of course). In this arrangement, each neuron on the first CONV layer has a 3x3 view of the input volume. A neuron on the second CONV layer has a 3x3 view of the first CONV layer, and hence by extension a 5x5 view of the input volume. Similarly, a neuron on the third CONV layer has a 3x3 view of the 2nd CONV layer, and hence a 7x7 view of the input volume. Suppose that instead of these three layers of 3x3 CONV, we only wanted to use a single CONV layer with 7x7 receptive fields. These neurons would have a receptive field size of the input volume that is identical in spatial extent (7x7), but with several disadvantages. First, the neurons would be computing a linear function over the input, while the three stacks of CONV layers contain non-linearities that make their features more expressive. Second, if we suppose that all the volumes have C channels, then it can be seen that the single 7x7 CONV layer would contain $C \times (7 \times 7 \times C) = 49C^2$ parameters, while the three 3x3 CONV layers would only contain $3 \times (C \times (3 \times 3 \times C)) = 27C^2$ parameters. Intuitively, stacking CONV layers with tiny filters as opposed to having one CONV layer with big filters allows us to express more powerful features of the input, and with fewer parameters. As a practical disadvantage, we might need more memory to hold all the intermediate CONV layer results if we plan to do backpropagation.

Recent departures. It should be noted that the conventional paradigm of a linear list of layers has recently been challenged, in Google's Inception architectures and also in current (state of the art) Residual Networks from Microsoft Research Asia. Both of these (see details below in case studies section) feature more intricate and different connectivity structures.

In practice: use whatever works best on ImageNet. If you're feeling a bit of a fatigue in thinking about the architectural decisions, you'll be pleased to know that in 90% or more of applications you should not have to worry about these. I like to summarize this point as "*don't be a hero*": Instead of rolling your own architecture for a problem, you should look at whatever architecture currently works best on ImageNet, download a pretrained model and finetune it on your data. You should rarely ever have to train a ConvNet from scratch or design one from scratch. I also made this point at the Deep Learning school.

Layer Sizing Patterns

Until now we've omitted mentions of common hyperparameters used in each of the layers in a ConvNet. We will first state the common rules of thumb for sizing the architectures and then follow the rules with a discussion of the notation:

The **input layer** (that contains the image) should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512.

The **conv layers** should be using small filters (e.g. 3x3 or at most 5x5), using a stride of $S = 1$, and crucially, padding the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input. That is, when $F = 3$, then using $P = 1$ will retain the original size of the input. When $F = 5$, $P = 2$. For a general F, it can be seen that $P = (F - 1)/2$ preserves the input size. If you must use bigger filter sizes (such as 7x7 or so), it is only common to see this on the very first conv layer that is looking at the input image.

The **pool layers** are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields (i.e. $F = 2$), and with a stride of 2 (i.e. $S = 2$). Note that this discards exactly 75% of the activations in an input volume (due to downsampling by 2 in both width and height). Another slightly less common setting is to use 3x3 receptive fields with a stride of 2, but this makes "fitting" more complicated (e.g., a 32x32x3 layer would require zero padding to be used with a max-pooling layer with 3x3 receptive field and stride 2). It is very uncommon to see receptive field sizes for max pooling that are larger than 3 because the pooling is then too lossy and aggressive. This usually leads to worse performance.

Reducing sizing headaches. The scheme presented above is pleasing because all the CONV layers preserve the spatial size of their input, while the POOL layers alone are in charge of down-sampling the volumes spatially. In an alternative scheme where we use strides greater than 1 or don't zero-pad the input in CONV layers, we would have to very carefully keep track of the input volumes throughout the CNN architecture and make sure that all strides and filters "work out", and that the ConvNet architecture is nicely and symmetrically wired.

Why use stride of 1 in CONV? Smaller strides work better in practice. Additionally, as already mentioned stride 1 allows us to leave all spatial down-sampling to the POOL layers, with the CONV layers only transforming the input volume depth-wise.

Why use padding? In addition to the aforementioned benefit of keeping the spatial sizes constant after CONV, doing this actually improves performance. If the CONV layers were to not zero-pad the inputs and only perform valid convolutions, then the size of the volumes would reduce by a small amount after each CONV, and the information at the borders would be "washed away" too quickly.

Compromising based on memory constraints. In some cases (especially early in the ConvNet architectures), the amount of memory can build up very quickly with the rules of thumb presented above. For example, filtering a 224x224x3 image with three 3x3 CONV layers with 64 filters each and padding 1 would create three activation volumes of size [224x224x64]. This amounts to a total of about 10 million activations, or 72MB of memory (per image, for both activations and gradients). Since GPUs are often bottlenecked by memory, it may be necessary to compromise. In practice, people prefer to make the compromise at only the first CONV layer of the network. For example, one compromise might be to use a first CONV layer with filter sizes of 7x7 and stride of 2 (as seen in a ZF net). As another example, an AlexNet uses filter sizes of 11x11 and stride of 4.

Case studies

There are several architectures in the field of Convolutional Networks that have a name. The most common are:

- **LeNet**. The first successful applications of Convolutional Networks were developed by Yann LeCun in 1990's. Of these, the best known is the LeNet architecture that was used to read zip codes, digits, etc.
- **AlexNet**. The first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet was submitted to the

ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).

- **ZF Net**. The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net). It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.
- **GoogLeNet**. The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google. Its main contribution was the development of an *Inception Module* that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. There are also several followup versions to the GoogLeNet, most recently Inception-v4.
- **VGGNet**. The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Their pretrained model is available for plug and play use in Caffe. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.
- **ResNet**. Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special *skip connections* and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network. The reader is also referred to Kaiming's presentation (video, slides), and some recent experiments that reproduce these networks in Torch. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice (as of May 10, 2016). In particular, also see more recent developments that tweak the original architecture from Kaiming He et al. Identity Mappings in Deep Residual Networks (published March 2016).

VGGNet in detail. Lets break down the VGGNet in more detail as a case study. The whole VGGNet is composed of CONV layers that perform 3x3 convolutions with stride 1 and pad 1, and of POOL layers that perform 2x2 max pooling with stride 2 (and no padding). We can write out the size of the representation at each step of the processing and keep track of both the representation size and the total number of weights:

```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,72
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,
6x128]  memory:  56*56*128=400K   weights: 0
```

```
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,64
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,29
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,29
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,29
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,29
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,29
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

As is common with Convolutional Networks, notice that most of the memory (and also compute time) is used in the early CONV layers, and that most of the parameters are in the last FC layers. In this particular case, the first FC layer contains 100M weights, out of a total of 140M.

Computational Considerations

The largest bottleneck to be aware of when constructing ConvNet architectures is the memory bottleneck. Many modern GPUs have a limit of 3/4/6GB memory, with the best GPUs having about 12GB of memory. There are three major sources of memory to keep track of:

- From the intermediate volume sizes: These are the raw number of **activations** at every layer of the ConvNet, and also their gradients (of equal size). Usually, most of the activations are on the earlier layers of a ConvNet (i.e. first Conv Layers). These are kept around because they are needed for backpropagation, but a clever implementation that runs a ConvNet only at test time could in principle reduce this by a huge amount, by only storing the current activations at any layer and discarding the previous activations on layers below.
- From the parameter sizes: These are the numbers that hold the network **parameters**, their gradients during backpropagation, and commonly also a step cache if the optimization is using momentum, Adagrad, or RMSProp. Therefore, the memory to store the parameter vector alone must usually be multiplied by a factor of at least 3 or so.
- Every ConvNet implementation has to maintain **miscellaneous** memory, such as the image data batches, perhaps their augmented versions, etc.

Once you have a rough estimate of the total number of values (for activations, gradients, and misc), the number should be converted to size in GB. Take the number of values, multiply by 4 to get the raw number of

bytes (since every floating point is 4 bytes, or maybe by 8 for double precision), and then divide by 1024 multiple times to get the amount of memory in KB, MB, and finally GB. If your network doesn't fit, a common heuristic to "make it fit" is to decrease the batch size, since most of the memory is usually consumed by the activations.

Additional Resources

Additional resources related to implementation:

- Soumith benchmarks for CONV performance
- ConvNetJS CIFAR-10 demo allows you to play with ConvNet architectures and see the results and computations in real time, in the browser.
- Caffe, one of the popular ConvNet libraries.
- State of the art ResNets in Torch7

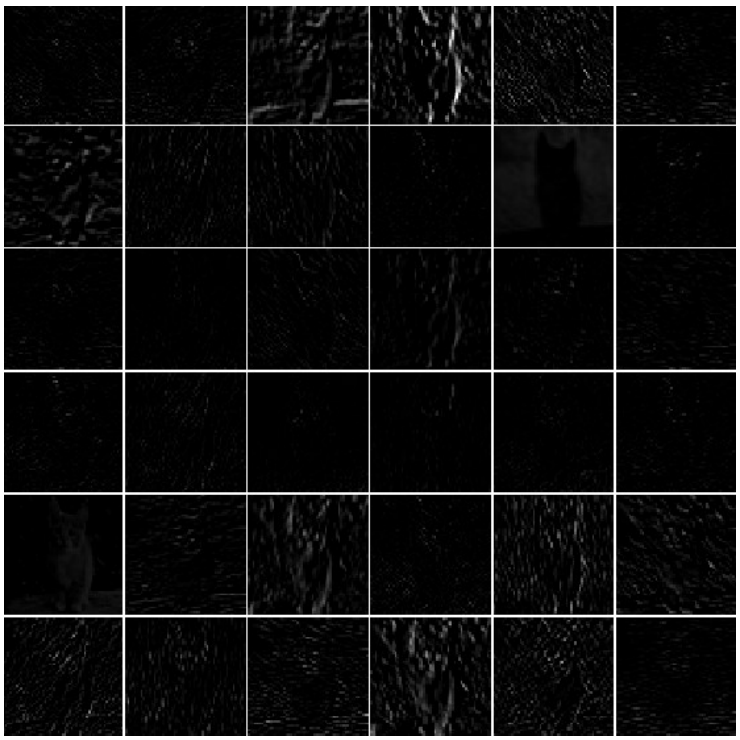
(this page is currently in draft form)

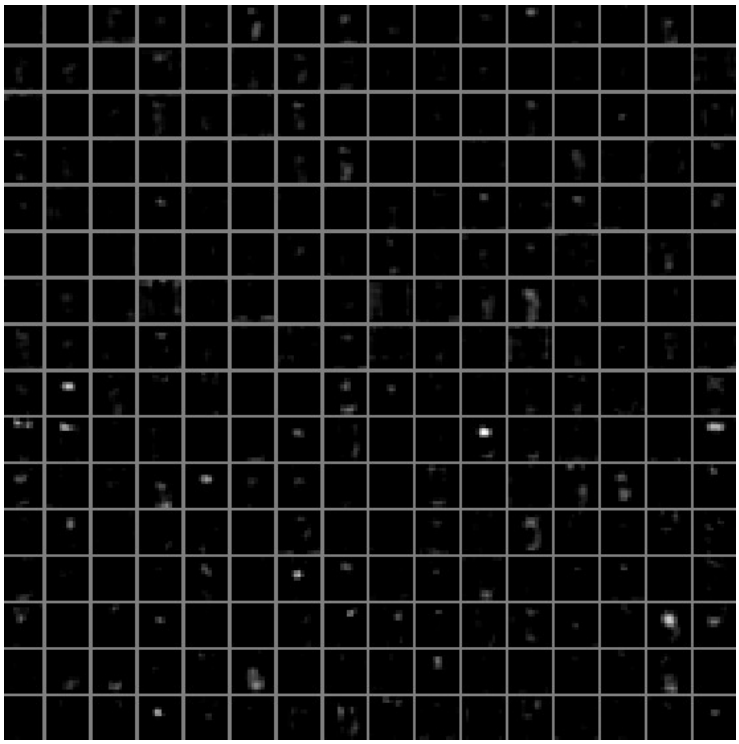
Visualizing what ConvNets learn

Several approaches for understanding and visualizing Convolutional Networks have been developed in the literature, partly as a response the common criticism that the learned features in a Neural Network are not interpretable. In this section we briefly survey some of these approaches and related work.

Visualizing the activations and first-layer weights

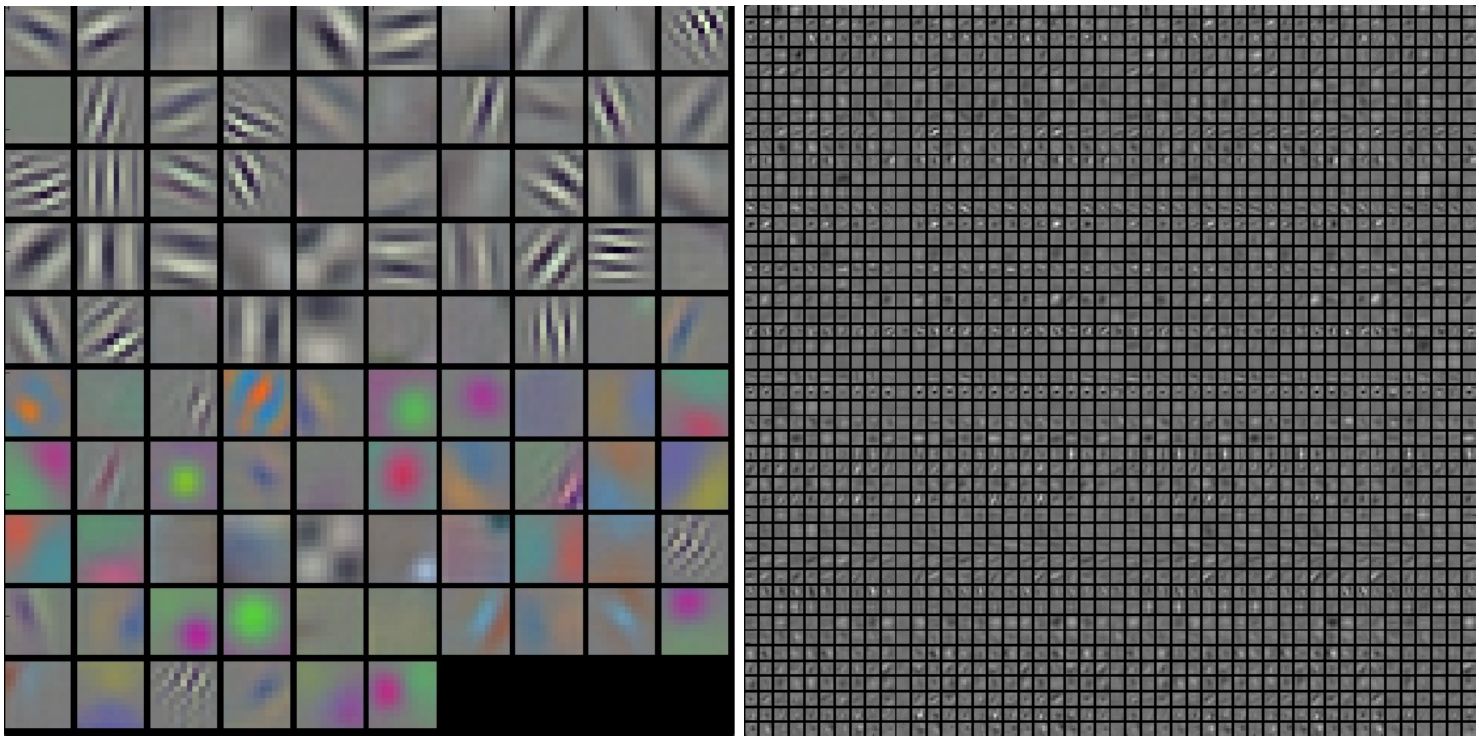
Layer Activations. The most straight-forward visualization technique is to show the activations of the network during the forward pass. For ReLU networks, the activations usually start out looking relatively blobby and dense, but as the training progresses the activations usually become more sparse and localized. One dangerous pitfall that can be easily noticed with this visualization is that some activation maps may be all zero for many different inputs, which can indicate *dead* filters, and can be a symptom of high learning rates.





Typical-looking activations on the first CONV layer (left), and the 5th CONV layer (right) of a trained AlexNet looking at a picture of a cat. Every box shows an activation map corresponding to some filter. Notice that the activations are sparse (most values are zero, in this visualization shown in black) and mostly local.

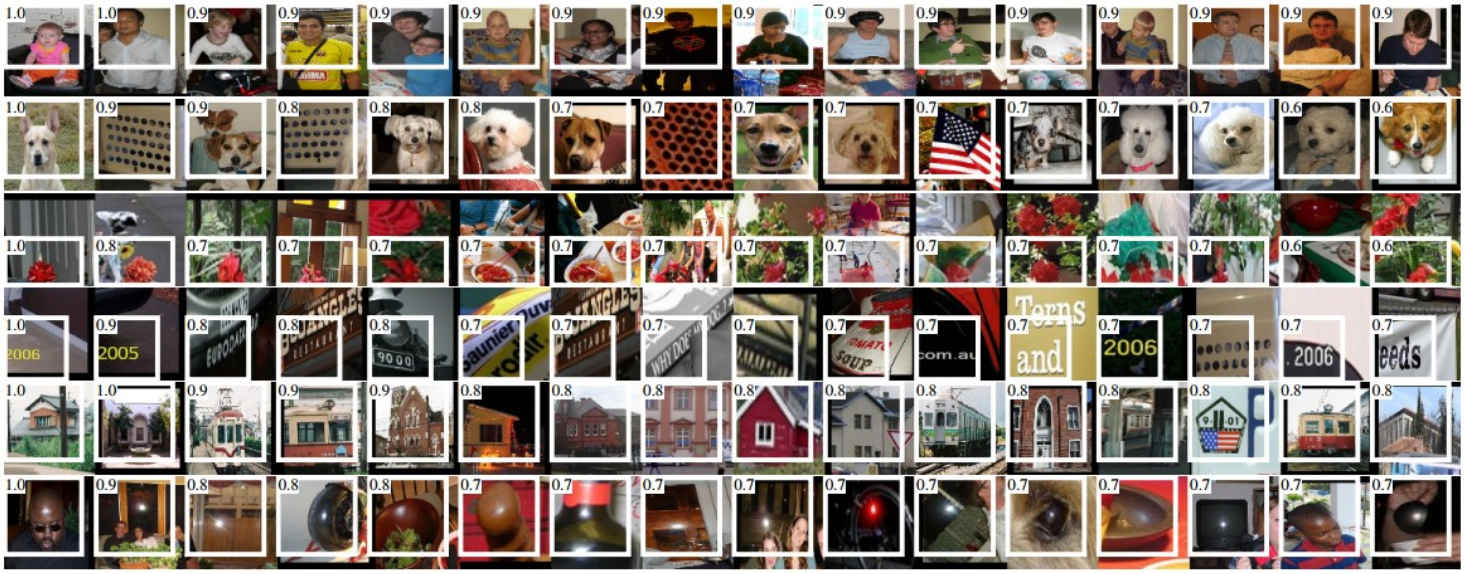
Conv/FC Filters. The second common strategy is to visualize the weights. These are usually most interpretable on the first CONV layer which is looking directly at the raw pixel data, but it is possible to also show the filter weights deeper in the network. The weights are useful to visualize because well-trained networks usually display nice and smooth filters without any noisy patterns. Noisy patterns can be an indicator of a network that hasn't been trained for long enough, or possibly a very low regularization strength that may have led to overfitting.



Typical-looking filters on the first CONV layer (left), and the 2nd CONV layer (right) of a trained AlexNet. Notice that the first-layer weights are very nice and smooth, indicating nicely converged network. The color/grayscale features are clustered because the AlexNet contains two separate streams of processing, and an apparent consequence of this architecture is that one stream develops high-frequency grayscale features and the other low-frequency color features. The 2nd CONV layer weights are not as interpretable, but it is apparent that they are still smooth, well-formed, and absent of noisy patterns.

Retrieving images that maximally activate a neuron

Another visualization technique is to take a large dataset of images, feed them through the network and keep track of which images maximally activate some neuron. We can then visualize the images to get an understanding of what the neuron is looking for in its receptive field. One such visualization (among others) is shown in Rich feature hierarchies for accurate object detection and semantic segmentation by Ross Girshick et al.:



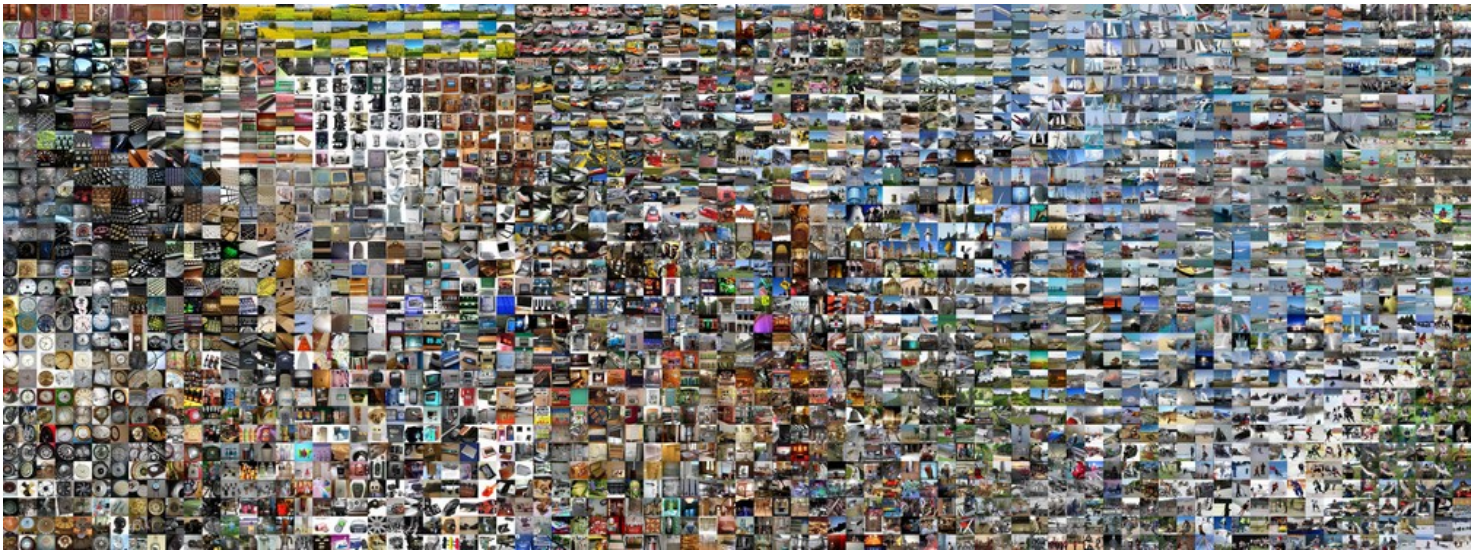
Maximally activating images for some POOL5 (5th pool layer) neurons of an AlexNet. The activation values and the receptive field of the particular neuron are shown in white. (In particular, note that the POOL5 neurons are a function of a relatively large portion of the input image!) It can be seen that some neurons are responsive to upper bodies, text, or specular highlights.

One problem with this approach is that ReLU neurons do not necessarily have any semantic meaning by themselves. Rather, it is more appropriate to think of multiple ReLU neurons as the basis vectors of some space that represents in image patches. In other words, the visualization is showing the patches at the edge of the cloud of representations, along the (arbitrary) axes that correspond to the filter weights. This can also be seen by the fact that neurons in a ConvNet operate linearly over the input space, so any arbitrary rotation of that space is a no-op. This point was further argued in Intriguing properties of neural networks by Szegedy et al., where they perform a similar visualization along arbitrary directions in the representation space.

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.

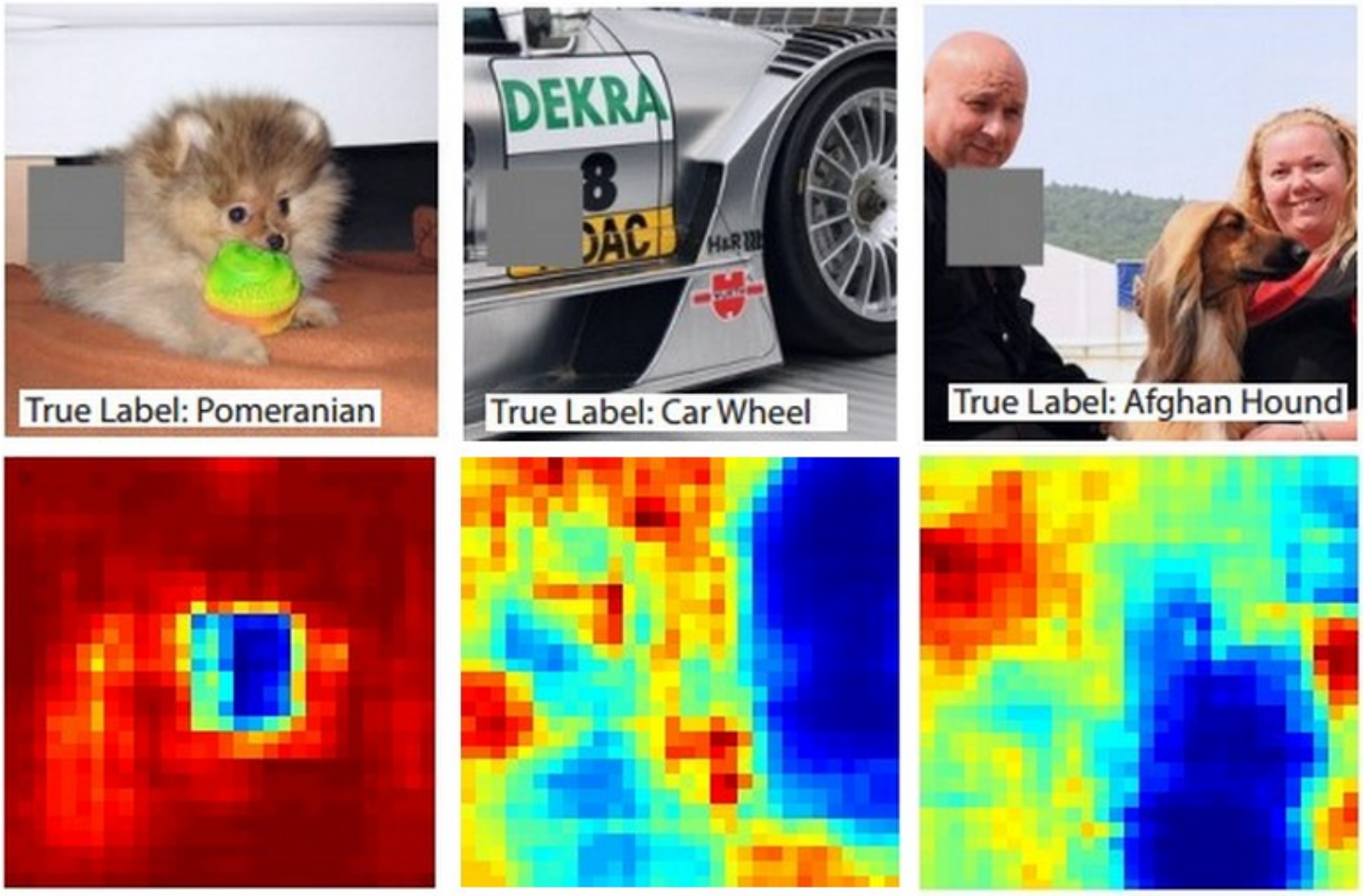
To produce an embedding, we can take a set of images and use the ConvNet to extract the CNN codes (e.g. in AlexNet the 4096-dimensional vector right before the classifier, and crucially, including the ReLU non-linearity). We can then plug these into t-SNE and get 2-dimensional vector for each image. The corresponding images can them be visualized in a grid:



t-SNE embedding of a set of images based on their CNN codes. Images that are nearby each other are also close in the CNN representation space, which implies that the CNN "sees" them as being very similar. Notice that the similarities are more often class-based and semantic rather than pixel and color-based. For more details on how this visualization was produced the associated code, and more related visualizations at different scales refer to t-SNE visualization of CNN codes.

Occluding parts of the image

Suppose that a ConvNet classifies an image as a dog. How can we be certain that it's actually picking up on the dog in the image as opposed to some contextual cues from the background or some other miscellaneous object? One way of investigating which part of the image some classification prediction is coming from is by plotting the probability of the class of interest (e.g. dog class) as a function of the position of an occluder object. That is, we iterate over regions of the image, set a patch of the image to be all zero, and look at the probability of the class. We can visualize the probability as a 2-dimensional heat map. This approach has been used in Matthew Zeiler's Visualizing and Understanding Convolutional Networks:



Three input images (top). Notice that the occluder region is shown in grey. As we slide the occluder over the image we record the probability of the correct class and then visualize it as a heatmap (shown below each image). For instance, in the left-most image we see that the probability of Pomeranian plummets when the occluder covers the face of the dog, giving us some level of confidence that the dog's face is primarily responsible for the high classification score. Conversely, zeroing out other parts of the image is seen to have relatively negligible impact.

Visualizing the data gradient and friends

Data Gradient.

Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps

DeconvNet.

Visualizing and Understanding Convolutional Networks

Guided Backpropagation.

Striving for Simplicity: The All Convolutional Net

Reconstructing original images based on CNN Codes

Understanding Deep Image Representations by Inverting Them

How much spatial information is preserved?

Do ConvNets Learn Correspondence? (tldr: yes)

Plotting performance as a function of image attributes

ImageNet Large Scale Visual Recognition Challenge

Fooling ConvNets

Explaining and Harnessing Adversarial Examples

Comparing ConvNets to Human labelers

What I learned from competing against a ConvNet on ImageNet

(These notes are currently in draft form and under development)

Table of Contents:

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. The three major Transfer Learning scenarios look as follows:

- **ConvNet as fixed feature extractor**. Take a ConvNet pretrained on ImageNet, remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the ConvNet as a fixed feature extractor for the new dataset. In an AlexNet, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier. We call these features **CNN codes**. It is important for performance that these codes are ReLUd (i.e. thresholded at zero) if they were also thresholded during the training of the ConvNet on ImageNet (as is usually the case). Once you extract the 4096-D codes for all images, train a linear classifier (e.g. Linear SVM or Softmax classifier) for the new dataset.
- **Fine-tuning the ConvNet**. The second strategy 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.
- **Pretrained models**. Since modern ConvNets take 2-3 weeks to train across multiple GPUs on ImageNet, it is common to see people release their final ConvNet checkpoints for the benefit of others

who can use the networks for fine-tuning. For example, the Caffe library has a Model Zoo where people share their network weights.

When and how to fine-tune? How do you decide what type of transfer learning you should perform on a new dataset? This is a function of several factors, but the two most important ones are the size of the new dataset (small or big), and its similarity to the original dataset (e.g. ImageNet-like in terms of the content of images and the classes, or very different, such as microscope images). Keeping in mind that ConvNet features are more generic in early layers and more original-dataset-specific in later layers, here are some common rules of thumb for navigating the 4 major scenarios:

1. *New dataset is small and similar to original dataset*. Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.
2. *New dataset is large and similar to the original dataset*. Since we have more data, we can have more confidence that we won't overfit if we were to try to fine-tune through the full network.
3. *New dataset is small but very different from the original dataset*. Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier form the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.
4. *New dataset is large and very different from the original dataset*. Since the dataset is very large, we may expect that we can afford to train a ConvNet from scratch. However, in practice it is very often still beneficial to initialize with weights from a pretrained model. In this case, we would have enough data and confidence to fine-tune through the entire network.

Practical advice. There are a few additional things to keep in mind when performing Transfer Learning:

- *Constraints from pretrained models*. Note that if you wish to use a pretrained network, you may be slightly constrained in terms of the architecture you can use for your new dataset. For example, you can't arbitrarily take out Conv layers from the pretrained network. However, some changes are straightforward: Due to parameter sharing, you can easily run a pretrained network on images of different spatial size. This is clearly evident in the case of Conv/Pool layers because their forward function is independent of the input volume spatial size (as long as the strides "fit"). In case of FC layers, this still holds true because FC layers can be converted to a Convolutional Layer: For example, in an AlexNet, the final pooling volume before the first FC layer is of size [6x6x512]. Therefore, the FC layer looking at this volume is equivalent to having a Convolutional Layer that has receptive field size 6x6, and is applied with padding of 0.
- *Learning rates*. It's common to use a smaller learning rate for ConvNet weights that are being fine-tuned, in comparison to the (randomly-initialized) weights for the new linear classifier that computes the class scores of your new dataset. This is because we expect that the ConvNet weights are relatively good, so we don't wish to distort them too quickly and too much (especially while the new Linear Classifier above them is being trained from random initialization).

Additional References

- CNN Features off-the-shelf: an Astounding Baseline for Recognition trains SVMs on features from ImageNet-pretrained ConvNet and reports several state of the art results.
- DeCAF reported similar findings in 2013. The framework in this paper (DeCAF) was a Python-based precursor to the C++ Caffe library.
- How transferable are features in deep neural networks? studies the transfer learning performance in detail, including some unintuitive findings about layer co-adaptations.
