# Lecture 2. Linear & Softmax Classifiers

Disclaimer: This note was modified from cs231n lecture notes by Prof. Li Fei-Fei at Stanford University.

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:



# 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. Morever, 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:

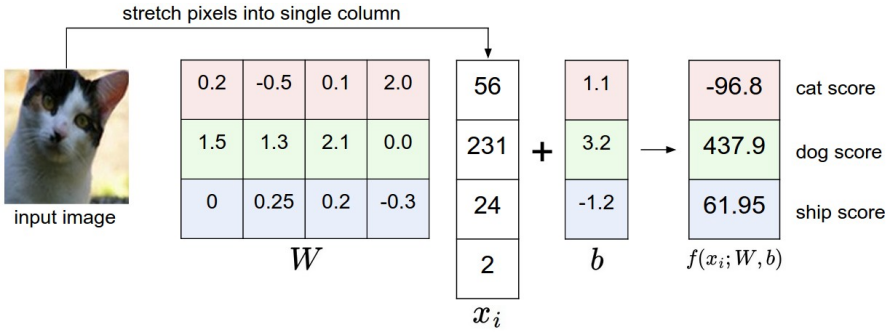
- First, note that the single matrix multiplication $Wx_i$ is effectively evaluating 10 separate classifiers in parallel (one for each class), where each classifier is a row of **W**.
- Notice also that we think of the input data $(x_i, y_i)$ as given and fixed, but we have control over the setting of 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).

---



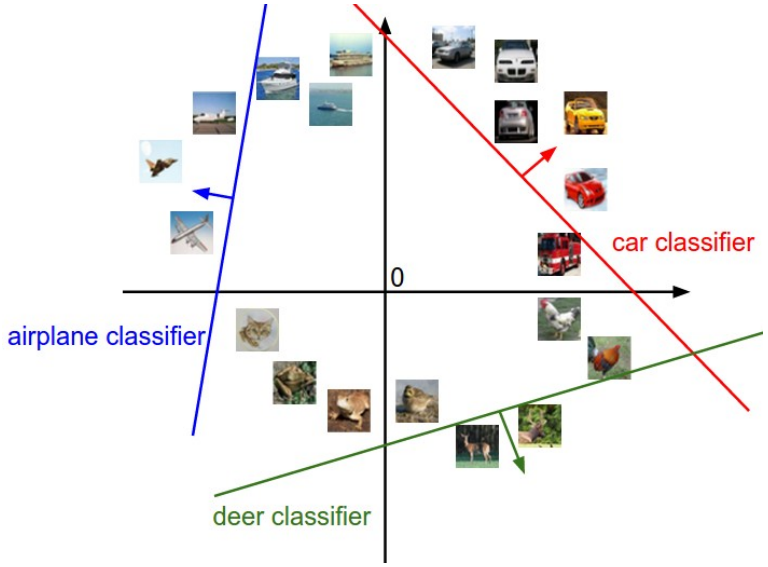

An example of mapping an image to class scores. For the sake of visualization, we assume the image only has 4 pixels (4 monochrome pixels, we are not considering color channels in this example for brevity), and that we have 3 classes (red (cat), green (dog), blue (ship) class). (Clarification: in particular, 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:

---

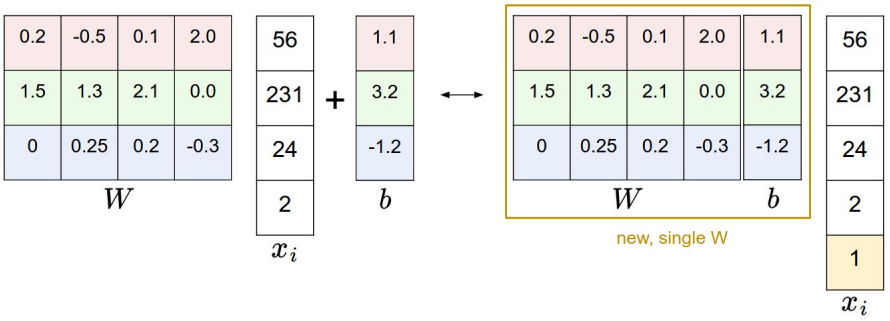



Illustration of the bias trick. Doing a matrix multiplication and then adding a bias vector (left) is equivalent to adding a bias dimension with a constant of 1 to all input vectors and extending the weight matrix by 1 column - a bias column (right). Thus, if we preprocess our data by appending ones 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 lecture, 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.

## Softmax classifier

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 **Softmax classifier**. If you've heard of the binary Logistic Regression classifier before, the Softmax classifier is its generalization to multiple classes. The Softmax classifier gives an 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 use 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 parameterized by $W$. To see this, remember 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:

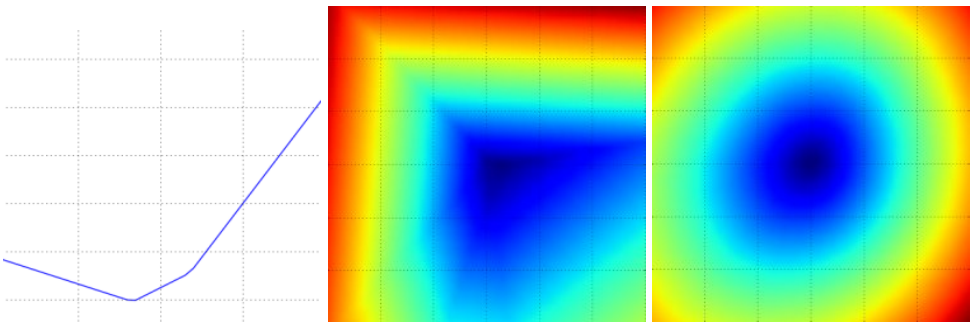
```
f = np.array([123, 456, 789]) # example with 3 classes and each having large scores
p = np.exp(f) / np.sum(np.exp(f)) # Bad: Numeric problem, potential blowup

# 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 *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.

## 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 hundred examples (right) in CIFAR-10. Left: one-dimensional loss by only varying **a**. Middle, Right: two-dimensional loss slice, Blue = low loss, Red = high loss. Notice the piecewise-linear structure of the loss function. The losses for multiple

examples are combined with average, so the bowl shape on the right is the average of many piece-wise linear bowls (such as the one in the middle).