

Introduction to Deep Learning

Hasan Ashraf

March 25, 2019

Contents

1	Image Classification	2
1.1	Nearest Neighbour	2
1.1.1	Distance Measures	2
1.2	k-Nearest Neighbours	3
1.2.1	Validation	4
1.2.2	Cross validation	5
1.3	Problems with Nearest Neighbour	5
1.3.1	Computational Complexity	5
1.3.2	Distance metrics are counter intuitive	5
2	Loss functions, Regularization	5
2.1	Template view	5
2.1.1	Score function	6
2.2	Data Loss Function	6
2.2.1	Hinge Loss - SVM	6
2.2.2	Cross entropy loss	6
2.3	Regularization	7
3	Matrix Derivatives	8
4	Neural Networks	9
4.1	Common Activation Functions	10
4.1.1	Sigmoid	10
4.1.2	tanh	11
4.1.3	ReLU	11
4.2	Representational Power	11

This script follows the introduction to deep learning lecture at TUM and also uses a great deal of material from Stanford's CS231n class along with other various sources that will be linked.

1 Image Classification

Some naive approaches for image classification:

1.1 Nearest Neighbour

The nearest neighbour is a very simple classifier. We take a test image (labeled data) and compute its distance from the training images. The test image then borrows its label from the closest training image.

As stated, this algorithm will have a complexity of $O(N)$ where N is the size of the training data. That is, each test image would require $O(N)$ of work because we would have to compare it to all images in the training set.

So if we have N training images and M test images, then we would end up doing $O(N * M)$ work.

1.1.1 Distance Measures

We can either use L1 or L2 distance as a distance measure. (Maybe we can employ any other p-norm? Although I don't know what sort of benefit that will accrue us.)

Note that the L1 distance is more resistant to bigger differences in the images while the L2 distance is less forgiving since we square the difference. Even though we later take the square root, the square root is taken over the sum of the squared differences. This is not a statement about the actual value of the L1 norm versus the L2 norm. Rather the contribution of a particular entry to the output.

Comment: This is what the Stanford website says. I think in order to see this, we would need to compute the contribution of a particular entry of the vector to the norm value. How might one do this?

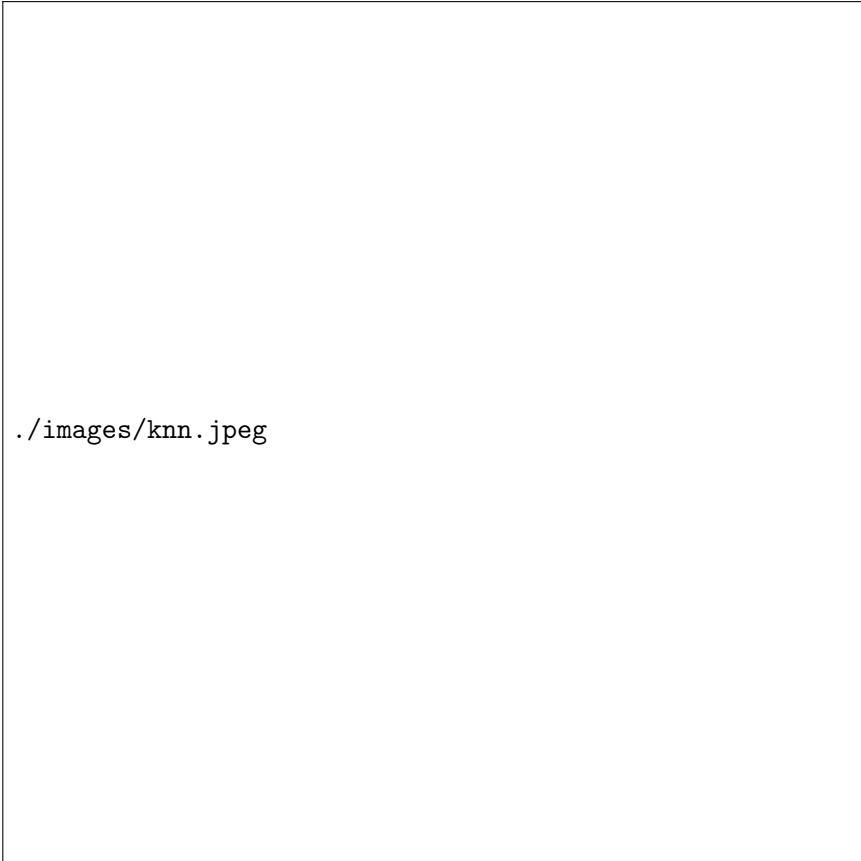
An intuitive explanation of this can be seen by looking at the graph of the sqrt function. What you will see is that larger values of x start pulling the norm to them. That is a vector with $[2,3,5,30]$ will have an L2 norm of 30.6 and an L1 norm of 40. What has happened here is that large difference in the two vectors has made the L2 norm concentrate around 30 since 30 squared is a much larger value than the sum of squares of the rest of the values in the vector.

It's unclear what, if anything, the significance of this would be at this stage. Perhaps we can come back to this later.

1.2 k-Nearest Neighbours

With k-Nearest Neighbours, instead of using the nearest neighbour to predict the label of an image, we poll the k-nearest neighbours and decide the output label in a democratic fashion. This gives us a much better estimate of the output label, depending on the value of the k.

So how do we choose k then? Larger values of k tend to smoothen decision boundaries, given of course that images with the same label lie in the same region of the vector space (approximately). As we go for higher values of k, the decision boundaries might shrink and we might get an increasing region of space for which we do not know which label we should use. On the other hand, smaller values of k might lead to islands of one label inside a larger spaces occupied by another. This discussion will make more sense if you take a look at the k-nearest neighbour figure



```
./images/knn.jpeg
```

1.2.1 Validation

Now to answer the question which we should answered in the last paragraph? Isn't that good writing practice? Anyway, k in this case is an example of a hyperparameter which we would like to choose. How do we choose hyperparameters you ask? First of all we divide our training set into two chunks: a new (large) training set and a small validation set. The validation set is meant to serve as a proxy test set which we can use to tune the hyperparameters. So we train our classifier on the new training set using different hyperparameters and then use the validation set to see which combination of hyperparameters gives us the best accuracy. Then we choose this value to to finally train our classifier and see what results we get on the actual test set.

The reason that we don't use the test set for validation is because we don't want to overfit our classifier to the test set. If we use the test set to perform validation then we would have no idea how well we might expect our classifier to behave out there. **cue wavy hand gesture to the beautiful fields outside. Apologies if all you can see are tall ugly buildings**

1.2.2 Cross validation

In some cases when our training set is small, we break it up into multiple folds. For example we can do a five fold decomposition of the training set and then for one set of validation parameters, we can alternatively training on 4 sets and validate on the 5th. We then vary the training and validation set and accumulate the accuracy values / other evaluation metric and average them over all such runs. Then we do this for another set of hyperparameter values. Of course this method is computationally expensive and is generally only used for small training sets.

1.3 Problems with Nearest Neighbour

1.3.1 Computational Complexity

As we discussed it, the nearest neighbour takes no time to train (just memorize the training set) but takes a lot of time to test. This is the opposite of what we want and what we usually achieve with neural networks.

1.3.2 Distance metrics are counter intuitive

In high dimensional spaces, distance metrics can be very counter intuitive since images that to our eyes are perceptually very different, can have the distance from each other. For example, take a look at this image from the Stanford course.

2 Loss functions, Regularization

2.1 Template view

One way to classify images is to build up a template from training data that we can then use to try to classify each image in the test data set. These templates are then encoded in a matrix "W" that allows us to compute class scores for each image. This is basically our score function with an added bias.

2.1.1 Score function

The score function for simple linear classifiers can just be a matrix with an added bias:

$$score(x_i) = W * x + b$$

Here $x \in R^n$ and W is an $m \times n$ matrix where m is the number of classes we want to classify.

In this view, each row of W is a template for the corresponding class. The reason we add a bias vector "b" is that we can interpret each row of W to be a plane in n dimensional space. Without the bias vector then, each plane would have to pass through the origin. This is not something we want of course and we translate the corresponding plane by the amount given by the b vector.

2.2 Data Loss Function

We want our score function to return us the right scores for most images in the training data set. It is important to keep in mind that the input data includes y_i , the class label for the i th input vector.

The data loss functions is a sort of penalty on misclassification. We use the result of our score function to compute the data loss function. Our goal is always to minimize the loss i.e. to get the right labels as often as we can. Two basic loss functions that we can use are

2.2.1 Hinge Loss - SVM

$$(1) \quad L_i = \sum_{j \neq y_i} \max(0, score[j] - score[y_i] + \delta)$$

The SVM loss function wants to put a distance of δ between the score for the right label and all of the wrong labels. In this view, we don't care about the magnitude of the score returned by the score function. Rather we only care about the relative ordering of things and the margins between the scores.

2.2.2 Cross entropy loss

The cross entropy loss comes from the softmax function which we can use to normalize the scores returned by our score function. While this comes from

information theory, I am not going to talk about the information theoretic interpretation of this since I don't really understand it yet.

However, here is a short description. The cross entropy between two distributions is given as:

$$(2) \quad H(p,q) = - \sum_x p(x) \log q(x)$$

Here p is the true distribution and q is some sort of an approximation. Now assume that x refers to the input label in this discussion and that our true distribution returns 1 for the true label and 0 for everything else. In this case the cross entropy reduces to $\log q(y_i)$ where y_i is the true label.

Now let's continue with the previous thread of discussion. Since we're normalizing the score values by the softmax function, we can interpret these new normalized values as being probabilities that the input belongs to a particular class. This is the view we will take in this document.

One more important thing to notice about the softmax function is that we take the exponentials of the raw scores. This again comes from the softmax function where we are interpreting the raw scores to be log probabilities and so we take the exponential to recover the "original value" back.

$$(3) \quad L_i = -\log \frac{e^{score[y_i]}}{\sum_j e^{score[j]}}$$

2.3 Regularization

In addition to minimizing the data loss function, we add a separate term to the overall loss function: regularization.

Assume that we have a particular matrix W that gives us minimal loss with the hinge loss function. A matrix $\alpha \times W$ would also satisfy this property given $\alpha \geq 1$. We don't want this since larger values in W tend to then overemphasize certain input dimensions and this does not generalize well -

i.e. we do not get good results on test data. Instead we want the weights to be diffused and take into account many different dimensions.

In order to penalize this, we minimize the sum of squares of the matrix entries:

$$(4) \quad \text{Regularization term} = \sum_j \sum_k W_{jk}^2$$

The overall loss function is then data loss + regularization.

3 Matrix Derivatives

For a matrix vector system with column vectors \vec{x} and \vec{y} :

$$\vec{y} = W \cdot \vec{x} \quad (5)$$

The various derivatives are given as follows:

$$(6) \quad \frac{\partial \vec{y}}{\partial \vec{x}} = W$$

For a row vector form of the same equation, we get the same thing. Note that this form has W transposed in the original system but the derivative is the transpose of the transposed W and we get W back.

Finally if we take the derivative of y with respect to W , the result will be a three dimensional array since y varies along one dimension, while W varies along two. However, the storage requirements are not as strict as this seems to imply since most of the values in this three dimensional array will be zero.

To see this, consider how y_1 depends on W .

$$(7) \quad y_1 = \sum_i W_{1,i} \cdot x_i$$

And the only partial derivatives of y with respect to elements of W that will be nonzero would be where y_i is differentiated with respect to $W_{i,j}$. That is the first index of W should be equal to the component of y we are taking the derivative with respect to.

The situation is opposite when we consider the row form of the same equation. Then we have to match the second component of W with the component of y under consideration.

Let us define the components of the three dimensional array that will hold the partial derivatives of components of y with respect to the components of W . Let's call this array F

$$F_{i,j,k} = \frac{\partial y_i}{\partial W_{j,k}} \quad (8)$$

Now, since the partial derivative is only non zero when the $i = j$, therefore, we can define a matrix G that contains the non zero partial derivatives:

$$G_{i,j} = W_{i,i,k} \quad (9)$$

4 Neural Networks

In our model, a neuron is nothing more than a function that takes in some input x , takes its dot product with weights w , adds a bias b and then optionally applies a nonlinear function to this output.

Biology: In terms of the biological inspiration, an artificial neuron gets multiple inputs at the synapses between its dendrites and the axons from other neurons. It integrates or sums these values up with certain weights w and depending on what its threshold is, it will then either fire or not.

Now a certain model from neuronal networks posits that information output from a neuron is encoded by its firing rate. The neuron gets inputs v_i from some other neuron, where v is the firing rate, and then weights this with some weight w_i . This gives us the input current for our neuron from neuron i .

If we do this for all inputs, then the output firing rate is given by:

$$\text{output firing rate} = g\left(\sum_j w_j \cdot v_j\right) \quad (10)$$

This is exactly how we define our artificial neural networks. g is what we usually call our activation function.

4.1 Common Activation Functions

4.1.1 Sigmoid

Since the sigmoid is basically a soft step function in the sense that we gradually ascend from 0 to 1, its derivative is almost zero when we get close to the extremes - or when our neuron saturates. During backpropagation then, this small, close to zero gradient, will make our output function agnostic to changes in the input that flow through the sigmoid neuron.

The other problem is that the sigmoid function is not zero centered. This has implications for the loss function as follows:

$$f = \sum w_i x_i + b \quad (11)$$

$$\frac{df}{dw_i} = x_i \quad (12)$$

$$\frac{dL}{dw_i} = \frac{dL}{df} \frac{df}{dw_i} = \frac{dL}{df} x_i \quad (13)$$

So the sign of the gradient always depends on the sign of the input. If in some particular gradient descent iteration, our gradient is positive or negative, depending on the input, then **all** the weights will increase or decrease. This is the only behaviour that is allowed: all weights will increase or decrease. This is what gives us the zigzagging behaviour during optimization.

4.1.2 tanh

The hyperbolic tangent function solves the zero-centered problem of the sigmoid but it still suffers from saturation on the edges.

4.1.3 ReLU

The rectified linear unit computes $\max(0, \text{output})$. We get a linear output from the ReLU if the input is greater than zero and otherwise we are stuck to zero.

x	y
-5	0
0	0
5	5
10	10

The ReLU function also suffers from the saturation problem that sigmoid and tanh neurons do since the gradient can end up being zero which means that the weights for the neuron cannot be updated anymore. This is more of a problem with ReLU neurons since they are thresholded to zero, while the sigmoid and tanh still have a derivative which is only close to zero and not identically zero. To solve this problem, sometimes we use leaky ReLU which has a small negative slope in -x region of the domain.

The ReLU function performs better than the sigmoid and tanh functions in general and can greatly speedup the learning of neural networks (convergence of gradient descent).

4.2 Representational Power

Neural networks can be used to approximate any continuous function f to some tolerance ϵ given enough neurons. This means that they have universal representational power. However this fact alone does not tell us much about what we can actually get from a neural network. For example, the indicator function:

$$I = \sum_j c_j \cdot 1 \tag{14}$$

Also has universal representational power. We can use multiple indicator functions which assume value c_i at the input x_i to approximate some function as well. (Think of the Dirac Delta function)