MLPR Tutorial Sheet 7 (Answers)

1. Learning a transformation:

The $K$-nearest-neighbour (KNN) classifier predicts the label of a feature vector by finding the $K$ nearest feature vectors in the training set. The label predicted is the label shared by the majority of the training neighbours. For binary classification, we normally choose $K$ to be an odd number so ties aren’t possible.

KNN is an example of a non-parametric method: no fixed-size vector of parameters (like $w$ in logistic regression) summarizes the training data. Instead, the complexity of the function represented by the classifier grows with the number of training examples $N$. Often, the whole training set needs to be stored so that it can be consulted at test time. (Gaussian processes are also a non-parametric method.)

a) How would the predictions from regularized linear logistic regression, with $p(y=1\mid x, w, b) = \sigma(w^\top x + b)$, and 1-nearest neighbours compare on the dataset below?

Answer:
Regularized linear regression would fit a decision boundary roughly along the line $x_2 = 1.1$. The confidence of its predictions depends on the size of regularization used. KNN does not give a straight decision boundary. In particular, it will classify points roughly around $(1.2, 1.3)$ and $(3, 1.3)$ as class 1, although those locations appear to be in the general band of class 0 points. The KNN classifications don’t come with a measure of uncertainty.

Non-parametric methods can have parameters. We could modify the KNN classifier by taking a linear transformation of the data $z = Ax$, and finding the $K$ nearest neighbours using the new $z$ features. One loss function for evaluating possible transformations $A$, could be the leave-one-out (LOO) classification error, defined as the fraction of errors made on the training set when the $K$ nearest neighbours for a training item may not include the point being classified. (It’s an $M$-fold cross-validation loss with $M = N$.)

b) Write down a matrix $A$ where the 1-nearest neighbour classifier has lower LOO error than using the identity matrix for the data above, and explain why it works.

Answer:
The transformation

$$A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix},$$

is one answer. Applying this matrix will throw away the horizontal $x_1$ feature, which appears irrelevant for classifying the points. Every training point will now be nearly on top of other training points with the same label, so the LOO error will be zero. Before the transformation, several points had a training point with the wrong class as their nearest neighbour.
c) Explain whether we can fit the LOO error for a KNN classifier by gradient descent on the matrix \( A \).

**Answer:**

An infinitesimal perturbation of almost all matrices \( A \) won’t change which points are the \( K \) nearest neighbours of any other points. Therefore all of the classification decisions will be the same, and the derivatives of the LOO score with respect to \( A \) are usually all zero. A gradient-based optimizer has no signal with which to adjust the transformation and will do nothing.

In edge cases, two neighbours could be nearly the same distance away, and a small change in \( A \) could change which of them is the nearest neighbour. The LOO score may change by a finite amount for an infinitesimal change, making the gradient undefined or infinite. So no, we can’t fit \( A \) with gradient methods.

For keen students: If we can create a loss function that’s a bit like the LOO error but differentiable, then we can fit \( A \) to that loss. For interest, that has been done: [https://papers.nips.cc/paper/2566-neighbourhood-components-analysis](https://papers.nips.cc/paper/2566-neighbourhood-components-analysis)

d) Assume that I have implemented some other classification method where I can evaluate a cost function \( c \) and its derivatives with respect to feature input locations: \( \bar{Z} \), where \( \bar{Z}_{nk} = \frac{\partial c}{\partial Z_{nk}} \) and \( Z \) is an \( N \times K \) matrix of inputs.

I will use that code by creating the feature input locations from a linear transformation of some original features \( Z = XA \). How could I fit the matrix \( A \)? If \( A \) is a \( D \times K \) matrix, with \( K < D \), how will the computational cost of this method scale with \( D \)?

You may quote results given in lecture note w7c.

**Answer:**

Using the rule for matrix products given in the lecture notes, we can backpropagate a derivative through a matrix product \( Z = XA \) with \( \bar{A} = X^\top \bar{Z} \). We now have the derivatives of the cost function with respect to the transformation matrix \( A \). We can use these derivatives in a gradient-based optimizer to fit \( A \).

There will probably be local optima, so we may wish to randomly initialize \( A \) multiple times to find a good local optimum.

We are taking \( D \)-dimensional features, reducing them to \( K \)-dimensions and then running the classifier. The only stage whose cost depends on \( D \) is the dimension-reducing step, as then the original data are discarded. Each of \( Z = XA \) and \( \bar{A} = X^\top \bar{Z} \) cost \( O(NDK) \). So any method with a classifier following a linear dimension reducing step should scale linearly with the original input dimensionality.

For keen students: Equation (5) in the NCA paper mentioned in the previous part costs \( O(D^2) \). Really keen students would work out how to fix the computation described in the paper to scale as \( O(D) \).

**Intended lessons of Q1:**

Anyone doing machine learning should know what KNN is. It can be a hard method to beat if you have a reasonable way to measure distance. An exam question could describe a method like KNN and ask you to compare it to methods that we have spent time on in the class: for example you might be expected to comment on complexity, storage requirements, and a comparison of the functions that the methods can
This exercise also gives practice in core parts of the course: thinking about linear transformations, and what they can do to data, and applying derivative rules that have been given to you: demonstrating you understand the high-level picture of reverse-mode differentiation and its consequences.

Finally, there is a big picture lesson for fitting models. If you want to learn one or two parameters, you can cross-validate choices made at random or on a grid. However, it’s hard to explore the space of many parameters (like a whole $A$ transformation matrix) that way, and gradient methods are often a good alternative. However, (standard) gradient-based optimizers need differentiable functions, and not all cost functions are differentiable.

For keen students only (non-examinable): the answers reference a research paper that has a clever trick for making a differentiable cost function. For comparison, taking a hard step function in a neural network and making it a differentiable sigmoid function means we can fit neural networks with gradient methods. Research on architectures like “neural Turing machines” and “memory networks”, and work surrounding the “concrete distribution”, are also driven by finding differentiable cost functions for interesting tasks.

### 2. Linear autoencoder

We centre our data so it has zero mean and fit a linear autoencoder with no bias parameters. The autoencoder is a $D$-dimensional vector-valued function $f$, computed from $D$-dimensional inputs $x$, using an intermediate $K$-dimensional “hidden” vector $h$:

$$h = W^{(1)} x,$$

$$f = W^{(2)} h.$$

Assume we want to find a setting of the parameters that minimizes the square error $\|f - x\|^2 = (f - x)^T (f - x)$, averaged (or summed) over training examples.

a) What are the sizes of the weight matrices $W^{(1)}$ and $W^{(2)}$? Why is it usually not possible to get zero error for $K < D$?

**Answer:**

$W^{(1)}$ is $K \times D$

$W^{(2)}$ is $D \times K$

$f = (W^{(2)} W^{(1)}) x$.

The geometry of this transformation was discussed in the answer to the maths background self-test. The final multiplication by $W^{(2)}$ brings points in $K$-dimensions up into $D$-dimensions, but the $\{f^{(n)}\}$ points will still all be in a $K$-dimensional linear subspace. Unless the $\{x^{(n)}\}$ points happen to lie exactly in a $K$-dimensional linear subspace, they can’t be fitted exactly.

An alternative explanation is that to get zero error, we need $W^{(2)} W^{(1)} = I$. However, this matrix product has rank at most $K$, and $I$ has rank $D$.

b) It’s common to transform a batch (or “mini-batch”) of data at one time. Given an
\(N \times D\) matrix of inputs \(X\), we set:

\[
H = XW^{(1)\top} \\
F = HW^{(2)\top}
\]

The total square error \(E = \sum_{nd}(F_{nd} - X_{nd})^2\), has derivatives with respect to the neural network output

\[
\frac{\partial E}{\partial F_{nd}} = 2(F_{nd} - X_{nd}), \text{ which we write as } \bar{F} = 2(F - X).
\]

Using the backpropagation rule for matrix multiplication,

\[
C = AB^\top \Rightarrow \bar{A} = \bar{CB} \text{ and } \bar{B} = \bar{C}^\top A,
\]

write down how to compute derivatives of the cost with respect to \(W^{(1)}\) and \(W^{(2)}\).

If time: you should be able to check numerically whether you are right.

**Answer:**

Applying the rule given in the question:

\[
E = \sum_{nd}(F_{nd} - X_{nd})^2 \quad \bar{F} = 2(F - X) \quad (2) \\
F = HW^{(2)\top} \quad H = FW^{(2)} \text{ and } \bar{W}^{(2)} = F^\top H \quad (3) \\
H = XW^{(1)\top} \quad \bar{W}^{(1)} = H^\top X \quad (4)
\]

The forwards computation goes from the bottom of the left column up to the top. Backpropagation then computes the second column from top to bottom. A demonstration of checking these expressions is available separately (probably not for discussion in tutorials, but you can ask questions online if not clear.)

c) The PCA solution sets \(W^{(1)} = V^\top\) and \(W^{(2)} = V\), where the columns of \(V\) contain eigenvectors of the covariance of the inputs. We only really need to fit one matrix to minimize square error.

Tying the weight matrices together: \(W^{(1)} = U^\top\) and \(W^{(2)} = U\), we can fit one matrix \(U\) by giving its gradients \(\bar{U} = \bar{W}^{(1)\top} + \bar{W}^{(2)}\) to a gradient-based optimizer.

Will we fit the same \(V\) matrix as PCA?

**Answer:**

I mainly put this question in just to show you an example of “parameter tying”. Sharing parameters in different parts of a model is common in machine learning. For example recent large word-based language models all use parameter tying, following https://arxiv.org/abs/1608.05859. It’s often easier to think about the parameters being separate at first, and then their derivatives just get added up.

The optimizer won’t set \(U = V\). The cost function only cares about how well the output matches the input, so the global optimum of the cost function is reached by setting \(U = VQ\), for any orthogonal matrix \(Q\). Then the function that is fitted is: \(f = (VQ)^\top V\top x = (VV^\top)x\), which matches what a PCA projection would do, and so has the same cost.

Any other reasonable argument is fine. For example, explaining that the hidden units can be permuted, and so there are multiple solutions.

While gradient-based optimizers do converge to the minimum of the cost function, I don’t think it’s totally straightforward to prove that they do. (I would welcome
a neat proof or reference to add here.) The cost function is not convex in $V$: an average of $V$ and $VQ$ does not give an optimal autoencoder.

3. Non-linear autoencoders

Some datapoints lie along the one-dimensional circumference of a semi-circle. You could create such a dataset, by drawing one of the features from a uniform distribution between $-1$ and $+1$, and setting the other feature based on that:

$$x_1^{(n)} \sim \text{Uniform}[-1, 1]$$
$$x_2^{(n)} = \sqrt{1 - (x_1^{(n)})^2}.$$

a) Explain why these points can’t be perfectly reconstructed when passed through the linear autoencoder in Q2 with $K=1$.

**Answer:**
The $\{f^{(n)}\}$ points must all lie in a $K$-dimensional linear subspace. With $K=1$, the outputs of the autoencoder all lie along a straight line in the 2-dimensional data space. The original data are on a line, but not a straight line, and so can’t be matched perfectly.

b) Explain whether the points could be perfectly reconstructed with $K=1$ by some non-linear decoder: $f = g(h)$. Where $g$ could be an arbitrary function, perhaps represented by multiple neural network layers. Assume the encoder is still linear: $h = W^{(1)}x$.

**Answer:**
We could set the weights so that $h = x_1$. Then set:

$$g_1(h) = h = x_1$$
$$g_2(h) = \sqrt{1 - h^2} = x_2,$$

thus perfectly reconstructing any input.

c) Explain whether the points could be perfectly reconstructed with $K=1$ by some non-linear encoder: $h = g(x)$. Where $g$ could again be an arbitrary function, perhaps represented by multiple neural network layers. Assume the decoder is still linear: $f = W^{(2)}h$.

**Answer:**
No matter how we set the non-linearity $g$ here, the outputs lie along a straight line, and so cannot match the original data.

Discussion of autoencoders (Q2 and Q3):

Autoencoders are often presented with tied weights, where the transpose of the “encoding” weights are used for “decoding”. That’s partly due to the link to PCA, and partly because of links to other models and initialization procedures. Such as Hinton and Salakhutdinov’s (2006) Science paper on deep learning.
However, it can make sense for encoding and decoding to look quite different. Non-linear functions can restore non-linear data from a small compact description. However, linear transformations may be flexible enough to find a good representation of high-dimensional vectors (they have $K \times D$ free parameters, which is a lot if $D$ is big!). I like linear transformations: they’re simple and topology preserving. Non-linear autoencoders can “tear” the input space, mapping nearby inputs into far off parts of the hidden space, and then reassemble them. Encoding with a linear transformation stops that from happening.

If you run out of things to do (I think most of the class actually have plenty to do), you could try to implement and fit some of the models mentioned above. For example, can you fit a dataset as well as PCA using the ideas in Q2? Or can you create and fit a dataset lying on a low-dimensional manifold as in Q3? There’s probably not time to discuss or debug code in your tutorial groups. However, I will comment on code posted to the forum:
```
# Put code between three backtics like this.
```

Tutorial 7 is the last tutorial, but you can still go to ML-Base to discuss your questions, questions in the notes, or past tutorial questions.