1. **Some computation with probabilities:** It’s common to compute with log-probabilities to avoid numerical underflow. Quick example: notice that the probability of 2000 coin tosses, \(2^{-2000}\), underflows to zero in Matlab, NumPy, or any package using IEEE 64 bit floating point numbers.

If we have two possible models, \(M_1\) and \(M_2\), for some features, we can define:

\[
\begin{align*}
    a_1 &= \log P(x \mid M_1) + \log P(M_1) \\
    a_2 &= \log P(x \mid M_2) + \log P(M_2).
\end{align*}
\]

Up to a constant, these ‘activations’ give the log-posterior probabilities that the model generated the features. Show that we can get the posterior probability of model \(M_1\) neatly with the logistic function:

\[
P(M_1 \mid x) = \sigma(a_1 - a_2) = \frac{1}{1 + \exp(-(a_1 - a_2))}.
\]

Now given \(K\) models, with \(a_k = \log [P(x \mid M_k) P(M_k)]\), show:

\[
\log P(M_k \mid x) = a_k - \log \sum_{k'} \exp a_{k'}.
\]

The \(\log \sum \exp\) function occurs frequently in the maths for probabilistic models (not just model comparison). Show that:

\[
\log \sum_k \exp a_k = \max_k a_k + \log \sum_k \exp \left( a_k - \max_{k'} a_{k'} \right).
\]

Explain why the expression is often implemented this way. (Hint: consider what happens when all the \(a_k\)’s are less than \(-1000\).

**Answer:**

Bayes’ rule: \(P(M_1 \mid x) = e^{a_1} / Z\) and \(P(M_2 \mid x) = e^{a_2} / Z\), where \(Z = P(x)\).

Only two possible models \(\Rightarrow P(M_1 \mid x) + P(M_2 \mid x) = 1\), \(Z = e^{a_1} + e^{a_2}\).

Substituting \(Z\),

\[
P(M_1 \mid x) = \frac{e^{a_1}}{e^{a_1} + e^{a_2}} = \frac{1}{1 + \exp(-a_1 - a_2)} = \sigma(a_1 - a_2).
\]

With \(K\) models \(P(x) = \sum_k P(x, M_k) = \sum_k e^{a_k}\), so Bayes’ rule gives:

\[
P(M_k \mid x) = \frac{e^{a_k}}{\sum_k e^{a_k}}, \quad \text{(a ‘softmax’)}
\]

\[
\log P(M_k \mid x) = a_k - \log \sum_{k'} e^{a_{k'}}.
\]

This expression involves \(e^{a_k}\)’s. If all the \(a_k\)’s are very negative, these could numerically underflow to zero, we’ll then take the log of zero and obtain -Inf.

We can multiply each of the terms in the sum by 1 = \(\exp(b) \exp(-b)\), for any \(b\):

\[
\log \sum_k \exp a_k = \log \left[ \exp(b) \sum_k \exp(a_k - b) \right]
= b + \log \sum_k \exp(a_k - b).
\]

---

1. Q2 is based on a previous sheet by Amos Storkey, Charles Sutton, and/or Chris Williams
The expression to be shown set \( b = \max_k a_k \). In this version, some of the terms might still underflow. However the largest term in the sum is now \( e^0 = 1 \). Any terms that underflow\(^2\) are less than \( \text{realmin} \approx 2 \times 10^{-308} \), and so even if we had used infinite precision, they would have had no practical effect when added on to a value \( \geq 1 \).

2. **Building a toy neural network**

Consider the following classification problem. There are two real-valued features \( x_1 \) and \( x_2 \), and a binary class label. The class label is determined by

\[
y = \begin{cases} 
1 & \text{if } x_2 \geq |x_1| \\
0 & \text{otherwise.}
\end{cases}
\]

(a) Can this function be perfectly represented by logistic regression, or a feedforward neural network without a hidden layer? Why or why not?

**Answer:**
No. A neural network without a hidden layer simply computes a linear function of the inputs. The decision boundary required for this problem is not linear.

(b) Consider a simpler problem for a moment, the classification problem

\[
y = \begin{cases} 
1 & \text{if } x_2 \geq x_1 \\
0 & \text{otherwise.}
\end{cases}
\]

Design a single ‘neuron’ that represents this function. Pick the weights by hand. Use the hard threshold function

\[
\Theta(a) = \begin{cases} 
1 & \text{if } a \geq 0 \\
0 & \text{otherwise},
\end{cases}
\]

applied to a linear combination of the \( x \) inputs.

**Answer:**
We could call the neuron \( h_1 \), and compute its output by

\[
h_1 = \Theta(W^{(1)}_{11} x_1 + W^{(1)}_{12} x_2 + b^{(1)}_1),
\]

and set

\[
W^{(1)}_{11} = -1, \quad W^{(1)}_{12} = 1, \quad b^{(1)}_1 = 0.
\]

Now \( z_1 > 0 \iff x_2 - x_1 \geq 0 \)

The notation for the parameters here is anticipating how the neuron will be used in the neural network in c). You might have called them something else, for example \( w_1, w_2, \) and \( b \).

(c) Now go back to the classification problem at the beginning of this question. Design a two layer feedforward network (that is, one hidden layer with two layers of weights) that represents this function. Use the hard threshold activation function as in the previous question.

---

\(^2\) The details of when and how numerical underflow happens come from the IEEE standard for floating point numbers.
Hints: Use two units in the hidden layer. The unit from the last question will be one of the units, and you will need to design one more. You can then find settings of the weights such that the output unit performs a binary AND operation on the two hidden units.

Answer:
We can define a second neuron $h_2$, 
\[ h_2 = \Theta(W_2^{(1)} x_1 + W_2^{(1)} x_2 + b_2^{(1)}) , \]
with 
\[ W_2^{(1)} = 1, \quad W_2^{(1)} = 1, \quad b_2^{(1)} = 0. \]
Now $z_2 > 0 \iff x_1 + x_2 \geq 0$. So $x_2 \geq -x_1$.
We can also write:
\[ h = \Theta(W^{(1)} x + b^{(1)}). \]
The intersection of the area for which $z_1 \geq 0$ and the area for which $z_2 \geq 0$ is the area where $y = 1$. We can set up a logical AND with the following output unit:
\[ f = \Theta(w_1^{(2)} h_1 + w_2^{(2)} h_2 + b^{(2)}) = \Theta(w^{(2)} \top h + b^{(2)}), \]
where we choose 
\[ w_1^{(2)} = 1, \quad w_2^{(2)} = 1, \quad b^{(2)} = -1.1 \]
You can show that $y = 1 \iff x_2 \geq |x_1|$. Check this relationship on a few example points to verify that it works.

Intended lessons of Q2:
This exercise is intended to step you through how using a hidden layer with a non-linearity can represent functions that linear functions can’t. The hard threshold or step function used in this question was common in early neural networks. It’s used here because it’s easier to reason about than a logistic sigmoid when building a toy network by hand.

3. 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 is sufficient to summarize the training data. The complexity of the function represented by the classifier can grow with the number of training examples $N$, but the whole training set needs to be stored so that it can be consulted at test time.

a) How would the predictions from regularized linear logistic regression, with 
\[ p(y=1 | x, w, b) = \sigma(w \top x + b) , \]
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 depend 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
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 w5a.

**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 Q3:**

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 we have spent more time on in the class: for example you might be expected to comment on complexity, storage requirements, and a comparison of the functions the methods can represent.

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. Current research on architectures like “neural Turing machines” and “memory networks” are also driven by finding differentiable cost functions for interesting tasks.