ANN (summer project), assignment 2

Vivek Kumar Singh

1 Theory Assignments:-

1. **Solution :**
(a) Linear regression as maximisation of likelihood function :
Let us assume that target variables and input variables are related as follows.

$$y^{(i)} = w^T x^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)}$ captures either unmodeled effects or random noise. Let us further assume that $\epsilon^{(i)}$ are distributed IID according to a Normal distribution with mean 0 and some σ^2.This can also be written as $''\epsilon^{(i)} \sim \mathcal{N}(0,\sigma^2)''$. i.e. density of $\epsilon^{(i)}$ is give by

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}\right)$$

This implies that

$$p(y^{(i)}|x^{(i)};w) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(y^{(i)} - w^T x^{(i)})^2}{2\sigma^2}\right)$$

This notation indicates the distribution $y^{(i)}$ given $x^{(i)}$ parameterised by w. We can also write this as $''y^{(i)}|x^{(i)}; w \sim \mathcal{N}(0,\sigma^2)''$.
The probability of the data is given by $p(\vec{y}|X;w)$ where X is the design matrix which contains all $x^{(i)}$'s. When we wish to view this as a function of w, we call it the likelihood function :

$$L(w) = p(\vec{y}|X;w)$$

. Note that by the independence assumption on the $\epsilon^{(i)}$'s (and hence also the $y^{(i)}$'s given the $x^{(i)}$'s), this can also be written

$$\begin{aligned} L(w) &= \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};w) \\ &= \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(y^{(i)} - w^T x^{(i)})^2}{2\sigma^2}\right) \end{aligned}$$

Instead of maximising $L(w)$, we can maximise any increasing function of it, in particular, we can instead maximize **log-likelihood** $\ell(w)$:

$$\begin{aligned}\ell(w) &= logL(w)\\ &= log\prod_{i=1}^{m}\frac{1}{\sqrt{2\pi}\sigma}exp\left(-\frac{(y^{(i)}-w^Tx^{(i)})^2}{2\sigma^2}\right)\\ &= \sum_{i=1}^{m}log\frac{1}{\sqrt{2\pi}\sigma}exp\left(-\frac{(y^{(i)}-w^Tx^{(i)})^2}{2\sigma^2}\right)\\ &= m\cdot log\frac{1}{\sqrt{2\pi}\sigma}-\frac{1}{\sigma^2}\cdot\frac{1}{2}\sum_{i=1}^{m}(y^{(i)}-w^Tx^{(i)})^2\end{aligned}$$

Hence maximising $\ell(w)$ gives the same answer as maximising

$$\frac{1}{2}\sum_{i=1}^{m}(y^{(i)}-w^Tx^{(i)})^2,$$

which we know as the least squares cost function $J(w)$.
To summarise : Under the probabilistic assumption, least squares cost function corresponds to finding the MLE of w.

(b) Instead of learning on outputs directly, we learn a probability distribution as this gives the probability of a point of having one class rather than with certainty classifying it as a class. As the probabilities will change continuously while learning, it will be a more accurate model than training on 0 and 1

2. **Solution :**
Stochastic Gradient Descent with momentum :
In this algorithm, we'll use the we'll combine SGD with momentum.
SGD implies, instead of using the whole batch of input data, we'll only use a single data point to update our parameters. This adds noise to the descent path and the cost function doesn't decrease monotonically, but we can see results pretty quickly and this is extremely useful in large scale ML application.
In momentum, introduces an additional term to remember what happened in the previous iteration. This memory dampens oscillations and smoothes out the gradient updates. We calculate exponentially weighted averages of the gradients and use them to update our the parameters instead of the usual gradients.

Implementation :
On iteration t :
Compute dW and db on the current data point
$v_{dW} = \beta v_{dW} + (1-\beta)dW$

$v_{db} = \beta v_{db} + (1-\beta)db$
$W = W - \alpha v_{dW}, b = b - \alpha v_{db}$
where β is a new hyperparameter and usually is between 0.8 and 0.99.
A default choice is $\beta = 0.9$.

This algorithm has some analogies with the momentum in physics. By using the running average, we are basically providing some initial momentum to the ball rolling down the hill of our convex function which makes it faster to reach the minimum point of the hill.

3. **Solution :**

   (a) **Mini-batchs :**
   Training NN with a large data is slow. So to find an optimisation algorithm that runs faster is a good idea. We split the dataset randomly into mini-batches of some fixed size (say 64) and run gradient descent on these mini-batches.
   Using batch GD takes too long per iteration. Using SGD is too noisy regarding the cost optimization, it will never converge and we loose all the speedup from vectorisation.
   On the other hand, using mini-batch GD provides faster learning as we still have the vectorisation advantage and it doesn't exactly converge but it oscillates in a very small region.

   (b) **Weight initialisation :**
   Symmetrical weight initialisation is a problem as if weights are symmetrical, all the neurons will also be symmetrical and will learn the same parameters. In the end, the whole NN will become equivalent to a single step.

   (c) **Overfitting :**
   If the hyperparameter λ is "good enough", regularisation will just reduce some weights that makes the neural network overfit.
   If λ good enough it will just make some of tanh activations roughly linear which will prevent overfitting.
   Other ways of regularising the model are :
   1. Using Dropout, in which we randomly knock out some units of the hidden layer so that our model can't rely on any one feature, so have to spread out weights.
   2. Data augmentation
   3. Early stopping
   4. Model Ensemble

   (d) **Batch Normalisation :**
   In batch normalisation, we normalise $z^{[l]}$ for each layer. The idea is the same as in normalising the inputs, making the NN run faster. This works because batch normalisation reduces the problem of input

values changing (shifting). It also adds some noise to $z^{[l]}$'s which create some regularisation effects.

4. **Solution :**
   Number of parameters $= 3 \cdot 3 \cdot 3 \cdot 8 = 216$
   Dimensions of output $= ((\frac{N_1 - 1}{2} + 1), (\frac{N_2 - 1}{2} + 1), 8)$