COMP9444: Neural Networks and Deep Learning

Week 3a. Backprop Variations

Alan Blair
School of Computer Science and Engineering
September 21, 2023

Outline

Cross Entropy

For *function approximation*, we normally use the *sum squared error* (SSE) loss:

$$E = \frac{1}{2}\sum_i (z_i - t_i)^2$$

where z_i is the output of the network, and t_i is the target output.

However, for *classification* tasks, where the target output t_i is either 0 or 1, it is more logical to use the *cross entropy* loss:

$$E = \sum_i \big(-t_i \log(z_i) - (1 - t_i)\log(1 - z_i)\big)$$

The motivation for these loss functions can be explained using the mathematical concept of *Maximum Likelihood*.

Maximum Likelihood (5.5)

Let H be a class of *hypotheses* for predicting observed *data* D.

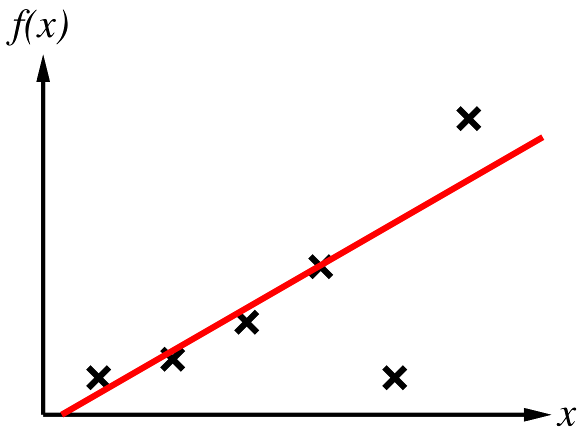
$\mathrm{Prob}(\mathrm{D}\,|\,h)$ = probability of data D being generated under hypothesis $h \in \mathrm{H}$.

$\log \mathrm{Prob}(\mathrm{D}\,|\,h)$ is called the *likelihood* of D, given h.

ML Principle: Choose $h \in \mathrm{H}$ which maximizes this likelihood,
i.e. maximize $\mathrm{Prob}(\mathrm{D}\,|\,h)$ [or, maximize $\log \mathrm{Prob}(D\,|\,h)$]

Here, the data D are the target values $\{t_i\}$ corresponding to input features $\{x_i\}$, and each hypothesis h is a function $f()$ determined by a neural network with specified weights or, to give a simpler example, $f()$ could be a straight line with a specified slope and y-intercept.

Least Squares Fit



Derivation of Least Squares

Due to the Central Limit Theorem, an accumulation of small errors will tend to produce "noise" in the form of a Gaussian distribution.

Suppose the data are generated by a linear function $f()$ plus Gaussian noise with mean zero and standard deviation σ. Then

$$\begin{aligned}
\mathrm{Prob}(\mathrm{D}|h) = \mathrm{Prob}(\{t_i\}|f) &= \prod_i \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(t_i - f(x_i))^2} \\
\log \mathrm{Prob}(\{t_i\}|f) &= \sum_i -\frac{1}{2\sigma^2}(t_i - f(x_i))^2 - \log(\sigma) - \frac{1}{2}\log(2\pi) \\
f_{ML} &= \text{argmax}_{f \in H} \log \mathrm{Prob}(\{t_i\}|f) \\
&= \text{argmin}_{f \in H} \sum_i (t_i - f(x_i))^2
\end{aligned}$$

(Note: we do not need to know σ)

Derivation of Cross Entropy (3.9.1)

For binary classification tasks, the target value t_i is either 0 or 1.

It makes sense to interpret the output $f(x_i)$ of the neural network as the *probability* of the true value being 1, i.e.

$$\begin{aligned}
P(1\,|\,f(x_i)) &= f(x_i) \\
P(0\,|\,f(x_i)) &= (1 - f(x_i)) \\
\text{i.e.} \quad P(t_i|\,f(x_i)) &= f(x_i)^{t_i}(1 - f(x_i))^{(1-t_i)}
\end{aligned}$$

$$\begin{aligned}
\log P(\{t_i\}|f) &= \sum_i t_i \log f(x_i) + (1 - t_i)\log(1 - f(x_i)) \\
f_{ML} &= \text{argmax}_{f \in H} \sum_i t_i \log f(x_i) + (1 - t_i)\log(1 - f(x_i))
\end{aligned}$$

(Can also be generalized to multiple classes.)

Cross Entropy and Backprop

Cross Entropy loss is often used in combination with sigmoid activation at the output node, which guarantees an output strictly between 0 and 1, and also makes the backprop computations a bit simpler, as follows:

$$\begin{aligned}
E &= \sum_i \big(-t_i \log(z_i) - (1 - t_i)\log(1 - z_i)\big) \\
\frac{\partial E}{\partial z} &= -\frac{t_i}{z_i} + \frac{1 - t_i}{1 - z_i} = \frac{z_i - t_i}{z_i(1 - z_i)}
\end{aligned}$$

If $z = \dfrac{1}{1 + e^{-s}}$, $\quad \dfrac{\partial E}{\partial s_i} = \dfrac{\partial E}{\partial z_i}\dfrac{\partial z_i}{\partial s_i} = z_i - t_i$

Cross Entropy and KL-Divergence

If we consider $p_i = \langle t_i, 1-t_i \rangle,\ q_i = \langle f(x_i), 1-f(x_i) \rangle$ as discrete probability distributions, the Cross Entropy loss can be written as:

$$
\begin{aligned}
\log P(\{t_i\}|\, f\,) &= \sum_i t_i \log f(x_i) + (1-t_i)\log(1-f(x_i)) \\
&= \sum_i \big[\big(t_i (\log f(x_i) - \log(t_i)) + (1-t_i)(\log(1-f(x_i)) - \log(1-t_i) \big) \\
&\qquad\qquad - \big(-t_i \log(t_i) - (1-t_i)\log(1-t_i) \big) \big] \\
&= \sum_i D_{\mathrm{KL}}(p_i \,\|\, q_i) \;-\; H(p_i)
\end{aligned}
$$

Since $H(p_i)$ is fixed, minimizing the Cross Entropy loss is the same as minimizing $\sum_i D_{\mathrm{KL}}(p_i \,\|\, q_i)$.

Cross Entropy and Outliers

SSE and Cross Entropy behave a bit differently when it comes to outliers.

SSE is more likely to misclassify outliers, because the loss function for each item is bounded between 0 and 1.

Cross Entropy is more likely to keep outliers correctly classified, because the loss function grows logarithmically (unbounded) as the difference between the target and network output approaches 1.

For this reason, Cross Entropy works particularly well for classification tasks that are *unbalanced* in terms of negative items vastly outnumbering positive ones (or vice versa).

Softmax (6.2.2)

- classification task with N classes
- neural network with N outputs $z_1, \ldots, z_N$
- assume the network's estimate for the probability of the correct class being j is proportional to $\exp(z_j)$
- because the probabilites must add up to 1, we need to *normalize* by dividing by their sum:

$$
\begin{aligned}
\mathrm{Prob}(i) &= \frac{\exp(z_i)}{\sum_{j=1}^N \exp(z_j)} \\
\log \mathrm{Prob}(i) &= z_i - \log \sum_{j=1}^N \exp(z_j)
\end{aligned}
$$

Log Softmax and Backprop

If the correct class is k, we can treat $-\log \mathrm{Prob}(k)$ as our cost function, and the gradient is

$$
\frac{d}{dz_i} \log \mathrm{Prob}(k) = \delta_{ik} - \frac{\exp(z_i)}{\sum_{j=1}^N \exp(z_j)} = \delta_{ik} - \mathrm{Prob}(i),
$$

where δ_{ik} is the *Kronecker delta*.

This gradient pushes up the correct class $i = k$ in proportion to the difference between its assigned probability and 1, and it pushes down the incorrect classes $i \neq k$ in proportion to the probabilities assigned to them by the network.

Softmax, Boltzmann and Sigmoid

If you have studied mathematics or physics, you may be interested to know that Softmax is related to the Boltzmann Distribution, with the negative of output z_i playing the role of the "energy" for "state" i.

The Sigmoid function can also be seen as a special case of Softmax, with two classes and one output, as follows:

Consider a simplified case where there is a choice between two classes, Class 0 and Class 1. We consider the output z of the network to be associated with Class 1 and we imagine a fixed "output" for Class 0 which is always equal to zero. In this case, the Softmax becomes:

$$\mathrm{Prob}(1) \;=\; \frac{e^z}{e^z + e^0} \;=\; \frac{1}{1+e^{-z}}$$

Weight Decay (5.2.2)

Sometimes we add a penalty term to the loss function which encourages the neural network weights w_j to remain small:

$$E = \frac{1}{2}\sum_i (z_i - t_i)^2 \;+\; \frac{\lambda}{2}\sum_j w_j^2$$

This can prevent the weights from "saturating" to very high values.

It is sometimes referred to as "elastic weights" because the weights experience a force as if there were a spring pulling them back towards the origin according to Hooke's Law.

The scaling factor λ needs to be determined from experience, or empirically.

Bayesian Inference

H is a class of hypotheses.

$\mathrm{Prob}(\mathrm{D}\,|\,h) =$ probability of data D being generated under hypothesis $h \in \mathrm{H}$.

$\mathrm{Prob}(h\,|\,\mathrm{D}) =$ probability that h is correct, given that data D were observed.

Bayes' Theorem:

$$\begin{aligned} \mathrm{Prob}(h\,|\,\mathrm{D})\mathrm{Prob}(\mathrm{D}) &= \mathrm{Prob}(\mathrm{D}\,|\,h)\,\mathrm{Prob}(h) \\ \mathrm{Prob}(h\,|\,\mathrm{D}) &= \frac{\mathrm{Prob}(\mathrm{D}\,|\,h)\,\mathrm{Prob}(h)}{\mathrm{Prob}(\mathrm{D})} \end{aligned}$$

$\mathrm{Prob}(h)$ is called the *prior* because it is our estimate of the probability of h *before* the data have been observed.

$\mathrm{Prob}(h\,|\,\mathrm{D})$ is called the *posterior* because it is our estimate of the probability of h *after* the data have been observed.

Weight Decay as MAP Estimation (5.6.1)

We assume a Gaussian prior distribution for the weights, i.e.

$$P(w) \;=\; \prod_j \frac{1}{\sqrt{2\pi}\sigma_0}\, e^{-w_j^2/2\sigma_0^2}$$

Then

$$P(w\,|\,t) = \frac{P(t\,|\,w)P(w)}{P(t)} = \frac{1}{P(t)}\prod_i \frac{1}{\sqrt{2\pi}\sigma}\, e^{-\frac{1}{2\sigma^2}(z_i - t_i)^2}\prod_j \frac{1}{\sqrt{2\pi}\sigma_0}\, e^{-w_j^2/2\sigma_0^2}$$

$$\log\, P(w\,|\,t) = -\frac{1}{2\sigma^2}\sum_i (z_i - t_i)^2 - \frac{1}{2\sigma_0^2}\sum_j w_j^2 \;+\; \mathrm{constant}$$

$$\begin{aligned} w_{\mathrm{MAP}} &= \text{argmax}_{w\in H}\; \log P(w\,|\,t) \\ &= \text{argmin}_{w\in H}\Big(\frac{1}{2}\sum_i (z_i - t_i)^2 + \frac{\lambda}{2}\sum_j w_j^2\Big), \quad \text{where } \lambda = \sigma^2/\sigma_0^2 \end{aligned}$$

This is known as Maximum A Posteriori (MAP) estimation.

Second Order Methods

Some optimization methods involve computing *second order* partial derivatives of the loss function with respect to each *pair* of weights:

$$\frac{\partial^2 E}{\partial w_i \partial w_j}$$

- Conjugate Gradients
  - approximate the landscape with a quadratic function (paraboloid) and jump to the minimum of this quadratic function
- Natural Gradients (Amari, 1995)
  - use methods from information geometry to find a "natural" re-scaling of the partial derivatives

These methods are not normally used for deep learning, because the number of weights is too high. In practice, the Adam optimizer tends to provide similar benefits with low computational cost.