Deep learning by Ian goodfellow

▼ Chapter 6 - Deep Feedforward Networks

- **Deep feedforward networks**, also called feedforward neural networks, or **multilayer perceptrons(MLPs)**, are the quintessential deep learning models.
- The goal of a feedforward network is to approximate some function $f*$. For example, for a classifier, $y=f*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y=f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation
- These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y.
- There are **no feedback connections** in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks.**
- The final layer of a feedforward network is called the **output layer.**
- Each example x is accompanied by a label $y \approx f*(x)$.The training examples specify directly what the output layer must do at each point x; it must produce a value that is close to y.
- The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of $f*$. Because the training data does not show the desired output for each of these layers, they are called **hidden layers.**

▼ 6.1 Learning XOR

- The XOR function (“exclusive or”) is an operation on two binary values, x1 and x2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0.

- The XOR function provides the target function y=f∗(x) that we want to learn.
- Our model provides a function y=f(x;θ), and our learning algorithm will adapt the parameters θ to make f as similar as possible to f∗.
- We can treat this problem as a regression problem and use a mean squared error loss function.

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\boldsymbol{x}\in\mathbb{X}} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta})\right)^2 .$$

- Specifically, we will introduce a simple feedforward network with one hidden layer containing two hidden units. See figure 6.2 for an illustration of this model.This feedforward network has a vector of hidden units h that are computed by a functionf(1)(x;W , c). The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to h rather than tox. The network now contains two functions chained together,h=f(1)(x;W , c) and y=f(2)(h;w, b), with the complete model being f(x;W , c, w, b) =f(2)(f(1)(x)).
- The activation functiong is typically chosen to be a function that is applied element-wise, with hi=g(x□W:,i+ci). In modern neural networks, the default recommendation is to use the rectified linear unit.

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$

▼ 6.2 Gradient Based Learning

- Designing and training a neural network is not much different from training any other machine learning model with gradient descent.
- The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become nonconvex.
- This means that neural networks are usuallyt rained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.
- For feedforward neural networks, it is important toinitialize all weights to small random values. The biases may be initialized to zero or to small positive values.
- As with other machine learning models, to apply gradient-based learning wemust choose a cost function, and we must choose how to represent the output ofthe model.

▼ 6.2.1 Cost functions

Fortunately, the cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.

In most cases, our parametric model defines a distributionp(y | x;θ) and we simply use the principle of maximum likelihood.

This means we use the cross-entropy between the training data and the model’s predictions as the cost function.

▼ Learning Conditional Distributions with Maximum Likelihood

Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data and the model distribution. This cost function is given by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}).$$

- An advantage of this approach of deriving the cost function from maximum
  likelihood is that it removes the burden of designing cost functions for each model.
  Specifying a model p(y | x) automatically determines a cost function log p(y | x).
- One recurring theme throughout neural network design is that the gradient of
  the cost function must be large and predictable enough to serve as a good guide for the learning algorithm.
- One unusual property of the cross-entropy cost used to perform maximum
  likelihood estimation is that it usually does not have a minimum value when applied to the models commonly used in practice.

▼ Learning Conditional Statistics

- Instead of learning a full probability distribution p(y | x; θ) we often want to learn just one conditional statistic of y given x.
- For example, we may have a predictor f(x; θ) that we wish to predict the mean
  of y.
- If we use a sufficiently powerful neural network, we can think of the neural
  network as being able to represent any function f from a wide class of functions, with this class being limited only by features such as continuity and boundedness rather than by having a specific

parametric form. From this point of view, we can view the cost function as being a functional rather than just a function.

- A functional is a mapping from functions to real numbers.
- We can thus think of learning as choosing a function rather than merely choosing a set of parameters.
- We can design our cost functional to have its minimum occur at some specific
  function we desire.

Our first result derived using calculus of variations is that solving the optimization problem

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2 \tag{6.14}$$

yields

$$f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y}\sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}], \tag{6.15}$$

- Unfortunately, mean squared error and mean absolute error often lead to poor
  results when used with gradient-based optimization. Some output units that
  saturate produce very small gradients when combined with these cost functions. This is one reason that the cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an
  entire distribution p(y | x).

▼ Output units

- The choice of cost function is tightly coupled with the choice of output unit.
- The choice of how to represent the output then determines the form of the cross-entropy function.
- Any kind of neural network unit that may be used as an output can also be
  used as a hidden unit.

- The role of the output layer is to provide some additional transformation from the features to complete the task that the network must perform.

▼ Linear Units for Gaussian Output Distributions

- One simple kind of output unit is an output unit based on an affine transformation with no nonlinearity

Given features $\boldsymbol{h}$, a layer of linear output units produces a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$.

- Linear output layers are often used to produce the mean of a conditional
  Gaussian distribution:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}).$$

- Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.
- The maximum likelihood framework makes it straightforward to learn the
  covariance of the Gaussian too, or to make the covariance of the Gaussian be a function of the input.
- It is difficult to satisfy such constraints with a linear
  output layer, so typically other output units are used to parametrize the covariance.
- Because linear units do not saturate, they pose little difficulty for gradient based optimization algorithms and may be used with a wide variety of optimization algorithms.

▼ Sigmoid Units for Bernoulli Output Distributions

- Many tasks require predicting the value of a binary variable y .
- Classification problems with two classes can be cast in this form.

- The maximum-likelihood approach is to define a Bernoulli distribution over y conditioned on x.
- A Bernoulli distribution is defined by just a single number.
- The neural net needs to predict only P(y = 1 | x). For this number to be a valid probability, it must lie in the interval [0, 1].
- Satisfying this constraint requires some careful design effort. Suppose we were to use a linear unit, and threshold its value to obtain a valid probability:

$$P(y = 1 \mid \boldsymbol{x}) = \max \left\{0, \min \left\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\right\}\right\} .$$

- This would indeed define a valid conditional distribution, but we would not be able to train it very effectively with gradient descent.
- Any time that (w^T)*h +b strayed outside the unit interval, the gradient of the output of the model with respect to its parameters would be 0.
- A gradient of 0 is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.
- Instead, it is better to use a different approach that ensures there is always a strong gradient whenever the model has the wrong answer.
- A sigmoid output unit is defined by:

$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right)$$

- We can think of the sigmoid output unit as having two components.
  - First, it uses a linear layer to compute z = (w ^T) h + b.

    - Next, it uses the sigmoid activation function to convert z into a probability.
- The sigmoid can be motivated by constructing an unnormalized probability distribution P˜(y), which does not sum to 1.
- We can then divide by an appropriate constant to obtain a valid probability distribution.
- If we begin with the assumption that the unnormalized log probabilities are linear in y and z, we can exponentiate to obtain the unnormalized probabilities.
- We then normalize to see that this yields a Bernoulli distribution controlled by a sigmoidal transformation of z:

$$\begin{aligned}\log \tilde{P}(y) &= yz \\ \tilde{P}(y) &= \exp(yz) \\ P(y) &= \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)} \\ P(y) &= \sigma\left((2y-1)z\right).\end{aligned}$$

- The z variable defining such a distribution over binary variables is called a logit.
- This approach to predicting the probabilities in log-space is natural to use
  with maximum likelihood learning
- Because the cost function used with maximum likelihood is − log P(y | x), the log in the cost function undoes the exp of the sigmoid.
- Without this effect, the saturation of the sigmoid could prevent gradient- based learning from making good progress.
- The loss function for maximum likelihood learning of a Bernoulli parametrized by a sigmoid is:

$$
\begin{aligned}
J(\boldsymbol{\theta}) &= -\log P(y \mid \boldsymbol{x}) \\
&= -\log \sigma\left((2y-1)z\right) \\
&= \zeta\left((1-2y)z\right).
\end{aligned}
$$

- By rewriting the loss in terms of the softplus function, we can see that it saturates only when (1 − 2y)z is very negative.
- As |z| becomes large while z has the wrong sign, the softplus function asymptotes toward simply returning its argument |z|
- The derivative with respect to z asymptotes to sign(z), so, in the limit of extremely incorrect z, the softplus function does not shrink the gradient at all.
- This property is very useful because it means that gradient-based learning can act to quickly correct a mistaken z.
- When we use other loss functions, such as mean squared error, the loss can saturate anytime σ(z) saturates. The sigmoid activation function saturates to 0 when z becomes very negative and saturates to 1 when z becomes very positive.
- For this reason, maximum likelihood is almost always the preferred approach to training sigmoid output units.
- Analytically, the logarithm of the sigmoid is always defined and finite, because the sigmoid returns values restricted to the open interval (0, 1), rather than using the entire closed interval of valid probabilities [0, 1]. In software implementations, to avoid numerical problems, it is best to write the negative log-likelihood as a function of z, rather than as a function of yˆ = σ(z). If the sigmoid function underflows to zero, then taking the logarithm of yˆ yields negative infinity.