Chapter 14

Machine learning

Machine Learning (ML) is a collective name for a number of techniques that approach problems we might consider 'intelligent', such as image recognition. In the abstract, such problems are mappings from a vector space of *feature*s, such as pixel values in an image, to another vector space of outcomes. In the case of image recognition of letters, this final space could be 26-dimensional, and a maximum value in the second component would indicate that a 'B' was recognized.

The essential characteristic of ML techniques is that this mapping is described by a – usually large – number of internal parameters, and that these parameters are gradually refined. The learning aspect here is that refining the parameters happens by comparing an input to both its predicted output based on current parameters, and the intended output.

14.1 Neural networks

The most popular form of ML these days is Deep Learning (DL), or *neural networks*. A neural net, or a deep learning network, is a function that computes a numerical output, given a (typically multi-dimensional) input point. Assuming that this output is normalized to the interval $[0, 1]$, we can use a neural net as a classification tool by introducing a threshold on the output.

Why 'neural'?

A neuron, in a living body, is a cell that 'fires', that is, gives off a voltage spike, if it receives certain inputs. In ML we abstract this to a *perceptron*: a function that outputs a value depending on certain inputs. To be specific, the output value is often a linear function of the inputs, with the result limited to the range $(0, 1)$.

14.1.1 Single datapoint

In its simplest form we have an input, characterized by a vector of *feature s* $\bar{x}$, and a scalar output y. We can compute y as a linear function of $\bar{x}$ by using a vector of *weights* of the same size, and a scalar *bias* b:

$$y = \bar{w}\bar{x} + b.$$

14.1.2 Sigmoid

To have some scale invariance, we introduce a function σ known as the *sigmoid* that maps $\mathbb{R} \to (0,1)$, and we actually compute the scalar output y as:

$$\mathbb{R} \ni y = \sigma(\bar{w}^t \bar{x} + b). \tag{14.1}$$

One popular choice for a sigmoid function is

$$\sigma(z) = \frac{1}{1+e^{-z}}.$$

This has the interesting property that

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

so computing both the function value and the derivative is not much more expensive than computing only the function value.

For vector-valued outputs we apply the sigmoid function in a pointwise manner:

```
// funcs.cpp
template <typename V>
void sigmoid_io(const V &m, V &a) {
    a.vals.assign(m.vals.begin(),m.vals.end());
    for (int i = 0; i < m.r * m.c; i++) {
        // a.vals[i]*=(a.vals[i]>0); // values will be 0 if negative, and equal to themselves
    if positive
        a.vals[i] = 1 / (1 + exp(-a.vals[i]));
    }
}
```

In other places (such as the findal layer of a DL network) a *softmax* function may be more appropriate.

14.1.3 Multi-dimensional output

It is rare that a single layer, defined by $\bar{w}, \bar{b}$ can achieve all that we ask of a neural net. Typically we use the output of one layer as the input for a next layer. This means that instead of a scalar y we compute a multi-dimensional vector $\bar{y}$.

Now we have weights and a bias for each component of the output, so

$$\mathbb{R}^n \ni \bar{y} = \sigma(W\bar{x} + \bar{b})$$

where W is now a matrix.

A few observations:

- As indicated above, the output vector typically has fewer components than the input, so the matrix is not square, in particular not invertible.
- The sigmoid function makes the total mapping non-linear.
- Neural nets typically have multiple layers, each of which is a mapping of the form $x \to y$ as above.

14.2 Deep learning networks

We will now present a full neural network. This presentation follows [105].

Use a network with $L \geq 1$ layers, where layer $\ell = 1$ is the input layer, and layer $\ell = L$ is the output layer.

For $\ell = 1, \ldots, L$, layer ℓ compute

$$\begin{aligned} z^{(\ell)} &= W^{(\ell)} a^{(\ell)} + b^{(\ell)} \\ y^{(\ell)} &= \sigma\big(y^{(\ell)}\big) \end{aligned} \tag{14.2}$$

where $a^{(1)}$ is the input and $z^{(L+1)}$ is the final output.

We write this compactly as

$$y^{(L)} = N_{\{W^{(\ell)}\}_\ell, \{b^{(\ell)}\}_\ell}\big(a^{(1)}\big) \tag{14.3}$$

where we will usually omit the dependence of the net on the $W^{(\ell)}, b^{(\ell)}$ sets.

```
// net.cpp
void Net::feedForward(const VectorBatch &input) {
    this->layers.front().forward(input); // Forwarding the input

    for (unsigned i = 1; i < layers.size(); i++) {
        this->layers.at(i).forward(this->layers.at(i - 1).activated_batch);
    }
}


// layer.cpp
void Layer::forward(const VectorBatch &prevVals) {

        VectorBatch output( prevVals.r, weights.c, 0 );
    prevVals.v2mp( weights, output );
        output.addh(biases); // Add the bias
    activated_batch = output;
    apply_activation<VectorBatch>.at(activation)(output, activated_batch);
}
```

14.2.1 Classification

In the above description both the input x and output y are vector-valued. There are also cases where a different type of output is desired. For instance, suppose we want to characterize bitmap images of digits; in that case the output should be an integer $0 \cdots 9$.

We accomodate this by letting the output y be in $\mathbb{R}^{10}$, and we say that the network recognizes the digit 5 if y_5 is sufficiently larger than the other output components. In this manner we keep the whole story still real-valued.

14.2.2 Error minimization

Often we have data points $\{x_i\}_{i=1,N}$ with known outputs y_i, and we want to make the network predict reproduce this mapping as well as possible. Formally, we seek to minimize the cost, or error:

$$C = \frac{1}{N} L(N(x_i), y_i)$$

over all choices $\{W\}, \{b\}$. (Usually we do not spell out explicitly that this cost is a function of all $W^{[\ell]}$ weight matrices and $b^{[\ell]}$ biases.)

```
float Net::calculateLoss(Dataset &testSplit) {
    testSplit.stack();
    feedForward(testSplit.dataBatch);
    const VectorBatch &result = output_mat();

    float loss = 0.0;
    for (int vec=0; vec<result.batch_size(); vec++) { // iterate over all items
      const auto& one_result = result.get_vector(vec);
      const auto& one_label  = testSplit.labelBatch.get_vector(vec);
      assert( one_result.size()==one_label.size() );
      for (int i=0; i<one_result.size(); i++) // Calculate loss of result
        loss += lossFunction( one_label[i], one_result[i] );
    }
    loss = -loss / (float) result.batch_size();

    return loss;
}
```

Minimizing the cost means to choose weights $\{W^{(\ell)}\}_\ell$ and biases $\{b^{(\ell)}\}_\ell$ such that for each x:

$$\left[\{W^{(\ell)}\}_\ell, \{b^{(\ell)}\}_\ell\right] = \operatorname*{argmin}_{\{W\},\{b\}} L(N_{\{W\},\{b\}}(x), y) \tag{14.4}$$

where $L(N(x), y)$ is a *loss function* describing the distance between the computed output $N(x)$ and the intended output y.

We find this minimum using *gradient descent*:

$$w \leftarrow w + \Delta w, \qquad b \leftarrow b + \Delta b$$

14.2.3 Coefficients computation

We are interested in partial derivatives of the cost wrt the various weights, biases, and computed quantities. For this it's convenient to introduce a short-hand:

$$\delta_i^{[\ell]} = \frac{\partial C}{\partial z_i^{[\ell]}} \qquad \text{for } 1 \le i \le n_\ell \text{ and } 1 \le \ell < L. \tag{14.5}$$

Now applying the chain rule we get (using $x \circ y$ for the pointwise (or *Hadamard*) vector-vector product $\{x_i y_i\}$):

- at the last level:

$$\delta^{[L-1]} = \sigma'\left(z^{[L-1]}\right) \circ \left(a^{[L]} - y\right)$$

- recursively for the earlier levels:

$$\delta^{[\ell]} = \sigma'\left(z^{[\ell]}\right) \circ \left(W^{[\ell+1]^t} \delta^{[\ell+1]}\right)$$

- sensitivity wrt the biases:

$$\frac{\partial C}{\partial b_i^{[\ell]}} = \delta_i^{[\ell]}$$

- sensitivity wrt the weights:

$$\frac{\partial C}{\partial w_{ik}^{[\ell]}} = \delta_i^{[\ell]} a_k^{[\ell-1]}$$

Using the special form

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

gives

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

14.2.4 Algorithm

We now present the full algorithm. Our network has layers $\ell = 1, \ldots, L$, where the parameter n_ℓ denotes the input size of layer ℓ.

Layer 1 has input x, and layer L has output y. Anticipating the use of minibatches, we let x, y denote a

group of inputs/output of size b, so their sizes are $n_1 \times b$ and $n_{L+1} \times b$ respectively.

$$
\begin{array}{lll}
\text{Input layer } \ell = 1 \text{ starts with:} & & \\
a^{(\ell)} = x & \text{network input} & n_{ell} \times b \\
\text{For layers } \ell = 1, \ldots, L & & \\
a^{(\ell)} & \text{layer input} & n_\ell \times b \\
W^{(\ell)} & \text{weights} & n_{\ell+1} \times n_\ell \\
b^{(\ell)} & \text{biases} & n_{\ell+1} \times b \\
z^{(\ell)} \leftarrow W^{(\ell)} a^{(\ell)} + b^{(\ell)} & \text{biased product} & n_{\ell+1} \times b \\
a^{(\ell+1)} = y^{(\ell)} \leftarrow \sigma\big(z^{(\ell)}\big) & \text{activated product} & n_{\ell+1} \times b \\
\text{The final output:} & & \\
y = a^{(L+1)} = z^{(L)} & & n_{L+1} \times b \\
\text{For layers } \ell = L, L-1, \ldots, 1 & & \\
D^{(\ell+1)} \leftarrow \mathrm{diag}\big(\sigma'(z^{(\ell)})\big) & & n_{\ell+1} \times n_{\ell+1} \\
\delta^{(\ell)} \leftarrow \begin{cases} D^{L+1}\big(a^{L+1} - y\big) \\ D^{(\ell+1)} {W^{(\ell+1)}}^t \delta^{(\ell+1)} \end{cases} \quad \ell < L & & n_{\ell+1} \times b \\
\Delta W^{(\ell)} \leftarrow \delta^{(\ell)} {a^{(\ell-1)}}^t & \text{weights update} & n_{\ell+1} \times ??? \\
w^{(\ell)} \leftarrow w^{(\ell)} - \Delta W^{(\ell)} & & \\
\Delta b^{(\ell)} \equiv \delta^{(\ell)} & \text{bias update} & n_{\ell+1} \times b
\end{array}
\tag{14.6}
$$

14.2.5 Computational aspects

Both in applying the net, the forward pass, and in learning, the backward pass, we perform *matrix times vector products*. This operation does not have much cache reuse, and will therefore not have high performance; section 1.7.11.

On the other hand, if we bundle a number of datapoints – this is sometimes called a *mini-batch* – and operate on them jointly, our basic operation becomes the *matrix times matrix product*, which is capable of much higher performance; section 1.6.1.2.

We picture this in figure 14.1:

- the input batch and output batch consist of the same number of vectors;
- the weight matrix W has a number of rows equal to the output size, and a number of columns equal to the input size.

We can now consider the efficient computation of $N(x)$. We already remarked that matrix-matrix multiplication is an important kernel, but apart from that we can use parallel processing. Figure 14.2 gives two parallelization strategies. In the first one, batches are divided over processes (or equivalently, multiple processes are working on independent batches simultaneously); in the second one, the input and output vector are themselves broken up.

Exercise 14.1. Consider the first scenario in a shared memory context. In this code:

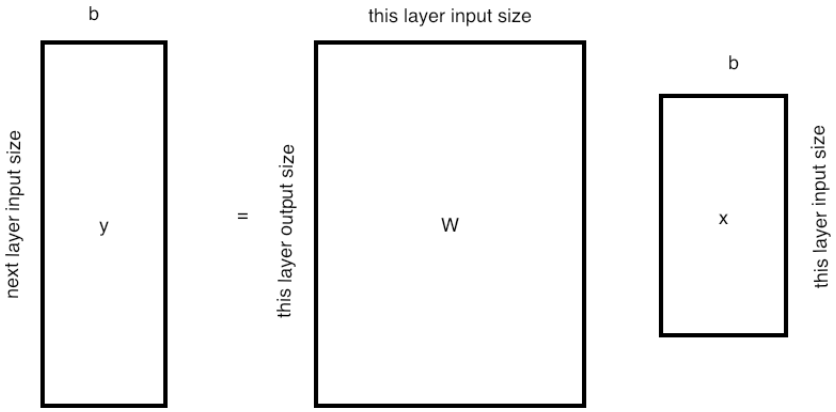

Figure 14.1: Shapes of arrays in a single layer

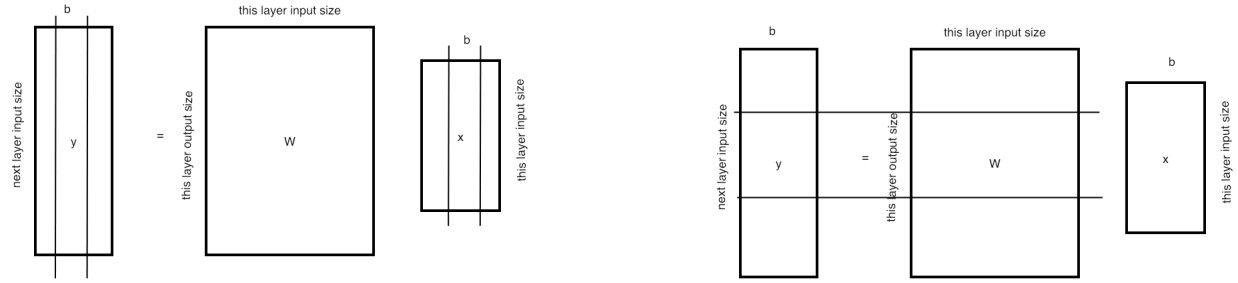

Figure 14.2: Partitioning strategies for parallel layer evaluation

for $b = 1, \ldots,$ batchsize
  for $i = 1, \ldots,$ outsize
    $y_{i,b} \leftarrow \sum_j W_{i,j} \cdot x_{j,b}$

assume that each thread computes part of the $1, \ldots,$ batchsize range.
Translate this to your favorite programming language. Do you store the input/output vectors by rows or columns? Why? What are the implications of either choice?

Exercise 14.2. Now consider the first scenario in a distributed memory context, with each process working on a slice (block column) of the batches. Do you see an immediate problem?

Exercise 14.3. Consider the second partitioning in a distributed memory context.

14.3 Stuff

Universal Approximation Theorem

Let $\varphi(\cdot)$ be a nonconstant,bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any function $f \in C(I_m)$ and $\varepsilon > 0$, there

exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi \left(w_i^T x + b_i \right)$$

as an approximate realization of the function f where f is independent of φ; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.