Chapter 1: Using Neural Nets to Recognize Handwritten Digits

In [4]:

```
%%javascript
MathJax.Hub.Config({
    TeX: { equationNumbers: { autoNumber: "AMS" }}
});
MathJax.Hub.Queue(
  ["resetEquationNumbers", MathJax.InputJax.TeX],
  ["PreProcess", MathJax.Hub],
  ["Reprocess", MathJax.Hub]
);
```

Perceptron: is a type of artificial neuron. It takes several binary inputs x_j (inputs that are either 0 or 1) and produces a single binary output.

For each input, there is a weight w_j associated with it, real numbers that express the importance of each input.

The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than a threshold value, which is a real number.

To simplify this expression, we can write weights and inputs as vectors and take their scalar product to represent the sum. The thresholds can be replaced by a bias $b = -\text{thresholds}$ so that the perceptron rule can be written as:

$$\text{output} = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases} \tag{1}$$

The bias can be thought of as a measure of how easy it is for the perceptron to "fire" or output a 1. The larger the b, the easier it is to fire.

The problem with a perceptron is its binary output. When we make a small change in weights or biases, we want a small change in output. However, for the perceptron, a small change in $\mathbf{w}$ or b may cause the output to flip discontinuously from 0 to 1 (or vice versa), affecting the rest of the network as well.

Sigmoid Neuron: is another type of artificial neuron. It takes in real number inputs x_j between $[0, 1]$, unlike the perceptron. It also has weights w_j associated with each input and a bias b.

The output of the sigmoid is determined by the sigmoid/logistic function $\sigma(z)$, which is defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{2}$$

where $z = \mathbf{w} \cdot \mathbf{x} + b$. The logistic/sigmoid function ensures that the output is a real number between 0 and 1, inclusive. This is in contrary to the output of the perceptron, which is simply a step function.

Such a function that governs the output of the neuron is called an **activation function**. The output value of an activation function is called the **activation value**.

The leftmost layer (first layer) of a neural network is called the **input layer** and the neurons within the layer are called **input neurons**.

The rightmost layer (last layer) of a neural network is called the **output layer** and the neurons within the layer are called **output neurons**.

The middle layers are called **hidden layers** simply because they are not input or output layers.

Neural networks in which the output from one layer is used as input for the next layer are called **feedforward neural networks**. There are no loops in such networks because information is always fed forward, never fed back.

Neural networs in which feedback loops are possible are called **recurrent neural networks (RNN)**. This is more closer to how our brains work.

Segmentation problem: splitting an image of many digits into a sequence of separate images, each containing a single digit.

To recognize digits, we use a three-layer neural network.

The input layer contains neurons encoding the value of the input pixels of an image.

Each scanned handwritten digit image consists of $28 \times 28 = 784$ pixels, so the input layer contains 784 input neurons.

Each input pixel is grayscale, with 0 representing white and 1 representing black, and various shades of gray in between $(0, 1)$.

The second layer of the network is a hidden layer with an arbitrary number of neurons, which will be fine-tuned later on.

The output layer of the network contains 10 neurons to represent the digits from 0 to 9. When classifying digits, the neural network decides on the output neuron with the highest activation value. For instance, if the 3rd output neuron (from 0th to 9th) has the highest activation value, much larger than the others, then it can confidently say that the input image is a 3.

The **MNIST** dataset contains 60000 training images and 10000 test images, each consisting of 784 grayscale pixels.

We denote a training input vector as $\mathbf{x}$, a 784-dimensional column vector. Each entry of the vector represents the grey value of a single pixel in the image.

The desired output vector is denoted by $\mathbf{y} = \mathbf{y}(\mathbf{x})$, which is a 10-dimensional column vector. For example, if an image $\mathbf{x}$ represents a 6, then $\mathbf{y}(\mathbf{x})$ would be

$$\mathbf{y}(\mathbf{x}) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$$

To obtain a measure of how much deviation the network's activation value predictions are from the actual value,

we define the **quadratic cost/loss/objective function** $C(\mathbf{W}, \mathbf{b})$, summed over all the images/training inputs $\mathbf{x}$ (also known as the **mean squared error MSE**):

$$C(\mathbf{W}, \mathbf{b}) = \frac{1}{2n} \sum_x \|\mathbf{y}(\mathbf{x}) - \mathbf{a}(\mathbf{x}, \mathbf{W}, \mathbf{b})\|^2 \tag{3}$$

The $1/2$ is introduced because we want the derivative of C to look pretty, without trailing a 2 in front, and the factor $1/n$ is introduced to normalize/scale the total cost by taking the average.

The cost function is non-negative because of the squares, and tends to 0 (becomes small) as the actual output vectors $\mathbf{a}$ tends to the desired output vectors $\mathbf{y}$.

Hence, the aim of our training algorithm will be to minimize the cost as a function of weights and biases.

We minimize the cost function instead of maximizing the number of images classified correctly because the cost function is a continuous function of the weights and biases and the number of images is not a continuous/smooth function.

We choose to use the MSE and minimize it because it is a standard measurement of error.

One way to minimize a function is to find the first derivative and set it to 0. This analytical method is VERY slow for large numbers of weights and biases. Another technique is called **gradient descent**, which is described as follows.

Suppose we have a function of two variables $C(v_1, v_2)$. A small change in C can be brought about by small changes in v_1 and v_2:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \tag{4}$$

We would like ΔC to be negative because we want to descend along a path which brings C to a minimum. That is, the updated C' should be smaller than the previous C:

$$C \to C' = C + \Delta C \tag{5}$$

The path of descent is chosen by choosing the values of Δv_1 and Δv_2 such that ΔC is negative

We define the vector of changes in v_i to be

$$\mathbf{\Delta v} = (\Delta v_1, \Delta v_2)^T$$

and the gradient of C to be

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T. \tag{6}$$

Hence, we can rewrite (4) as

$$\Delta C \approx \nabla C \cdot \Delta \mathbf{v}. \tag{7}$$

If we want ΔC to be negative, we choose

$$\Delta \mathbf{v} = -\eta \nabla C, \tag{8}$$

where η is a positive parameter known as the **learning rate**, chosen within the limits of the approximation in (5) to ensure that ΔC always stays negative: $\Delta C \approx -\eta ||\nabla C||^2$. We use (6) to compute a value for $\Delta \mathbf{v}$ and update the vector $\mathbf{v}$ by that incremental amount:

$$\mathbf{v} \rightarrow \mathbf{v}' = \mathbf{v} - \eta \nabla C \tag{9}$$

Incrementally updating the vector like this draws us a path of descent for C and hence, C keeps decreasing until we reach a **global minimum** (or a **local minimum** if we are not so lucky).

In summary, the gradient descent algorithm is a procedure to repeatedly compute the gradient ∇C, and move in the negative/opposite direction of it, "falling down" the slope of a valley. The whole argument above can be applied to a cost function with any number of variables.

If we try to mimic a rolling ball for gradient descent, we run into second partial derivatives, which require exponentially more computations, making it impractical to use. However, there are other methods that avoid this problem. (Perhaps Hamiltonian mechanics could be used?)

The problem with gradient descent is that we try to compute the cost fuction gradient separately for each training input $\mathbf{x}$ and average them, which is slow when there are a large number of training inputs.

To overcome this problem, we use the method of **stochastic gradient descent**. The idea is to randomly pick out a small sample of training inputs, compute the average cost gradient, update the variable vector and incrementally decrease the cost, and repeat the procedure with another small sample until we have exhausted all the training data. This is said to complete an **epoch** of training. At this point, we start over with a new training epoch.

Stochastic gradient descent speeds up training because we are not calculating the average cost gradient for the full batch size of $n = 60000$ inputs and then updating our variable vector and cost afterwards, and repeating. Instead, we calculate the average cost gradient for a small mini-batch of $m = 10$ inputs, update the variable vector and cost, and repeating. This speeds up gradient estimation by a factor of 6000.

Think about gradient descent like a sober man carefully walking in small steps on the shortest path to his destination. Compare this to stochastic gradient descent, which is like a drunk man randomly taking large steps towards the general direction of his destination. The exact computation of the gradient is not important, we need only to move in a general direction towards the minima to decrease the cost C.

Chapter 2: How the Backpropagation Algorithm Works

Backpropagation: is a fast algorithm for computing the gradient of the cost function. This allows our neural

network to learn faster from the gradient descent algorithm. Backpropagation is about understanding how changing the weights and biases in a network changes the cost function.

The weight of the connection from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer is denoted by w^l_{jk}. Bias of the j^{th} neuron in the l^{th} layer is denoted by b^l_j. Activation of the j^{th} neuron in the l^{th} layer is denoted by a^l_j.

$$\text{weight } w^l_{jk} : \text{connects } k^{\text{th}} \text{ neuron in } (l-1)^{\text{th}} \text{ layer } \rightarrow j^{\text{th}} \text{ neuron in } l^{\text{th}} \text{ layer}$$

$$\text{bias } b^l_j : \text{bias of } j^{\text{th}} \text{ neuron in } l^{\text{th}} \text{ layer}$$

$$\text{activation } a^l_j : \text{activation of } j^{\text{th}} \text{ neuron in } l^{\text{th}} \text{ layer}$$

With this notation, we can write the activation a^l_j of the j^{th} neuron in the l^{th} layer to be related to the activation a^{l-1}_k of the k^{th} neuron in the $(l-1)^{\text{th}}$ layer through the equation:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j \right) \tag{10}$$

Equation (10) is written in components, which represent the activation and bias vectors and the weight matrix. We write it in a compact, vectorized form, without indices, also defining the **weighted input** $z^l_j = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$ to the activation function for neuron j in layer l:

$$\mathbf{a}^l = \sigma(\mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l) \tag{11}$$
$$\mathbf{a}^l = \sigma(\mathbf{z}^l)$$

The goal of backpropagation is to compute the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ of the cost function C with respect to any weight w or bias b in the network.

For backpropagation to work, we need to make two main assumptions about the form of the cost function. The cost function is written as an average:

$$C = \frac{1}{2n} \sum_{\mathbf{x}} \|\mathbf{y}(\mathbf{x}) - \mathbf{a}^L(\mathbf{x})\|^2, \tag{12}$$

where n is the total number of training inputs, the sum is over individual training inputs $\mathbf{x}$, $\mathbf{y} = \mathbf{y}(\mathbf{x})$ is the corresponding desired output, L is the number of layers in the network (so the L^{th} layer is the last layer in the network), and $\mathbf{a} = \mathbf{a}^L(\mathbf{x})$ is the output activation vector of the last layer in the network when $\mathbf{x}$ is input.

The first assumption about the cost function is that it can be written as an average $C = \frac{1}{n} \sum_{\mathbf{x}} C_{\mathbf{x}}$ over cost functions $C_{\mathbf{x}}$ for individual training inputs $\mathbf{x}$.

The reason for this assumption is because we want to compute the partial derivatives $\partial C_{\mathbf{x}}/\partial w$ and $\partial C_{\mathbf{x}}/\partial b$ for singular training inputs and then recover $\partial C/\partial w$ and $\partial C/\partial b$ by averaging over training inputs.

The second assumption about the cost function is that it can be written as a function of the activation output values from the last layer in the neural network. For instance, for a single training input $\mathbf{x}$, the cost function can be written as:

$$C_{\mathbf{x}} = \frac{1}{2}\|\mathbf{y} - \mathbf{a}^L\|^2 = \frac{1}{2}\sum_i (y_i - a_i^L)^2, \tag{13}$$

where i stands for the index number of neurons in the last L^{th} layer of the network.

Hadamard product: is the elementwise product of two vectors of the same length/dimension. For instance, for vectors $\mathbf{s}$ and $\mathbf{t}$, the Hadamard product between them is written as:

$$\mathbf{s} \odot \mathbf{t} = (s \odot t)_j = s_j t_j \tag{14}$$

We introduce an intermediate quantity δ_j^l, which is the **error** of the j^{th} neuron in the l^{th} layer. This error is defined in the following way:

As the weighted input z_j^l comes in to this neuron, instead of outputting $\sigma(z_j^l)$, the neuron instead outputs $\sigma(z_j^l + \Delta z_j^l)$. This change propagates throughout the network and so the overall change in cost is given by:

$$\Delta C = \frac{\partial C}{\partial z_j^l}\Delta z_j^l,$$

where Δz_j^l is the change in weighted input. This change is such that in updating $C \to C' = C + \Delta C$, the updated cost is smaller than the previous cost, so Δz_j^l has the opposite sign to $\partial C/\partial z_j^l$. The closer $\partial C/\partial z_j^l$ is to 0, the smaller the perturbation Δz_j^l needs to be, so a good measure of the error in the neuron can be given by:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{15}$$

The vector δ^l denotes the vector of errors for all neurons associated with layer l.

Equation for the error in the output layer:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{16}$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

The first term $\partial C/\partial a_j^L$ measures how fast the cost is changing as a function of the j^{th} neuron's output activation in the last L^{th} layer. The second term $\sigma'(z_j^L)$ measures how fast the activation function σ is changing at z_j^L. Equation (15) is a componentwise expression, so we can rewrite it in matrix-based form:

$$\delta^L = \nabla_a C \odot \sigma'(\mathbf{z}^L), \tag{17}$$

where $\nabla_a C$ is defined to be a vector whose components are the partial derivatives $\partial C/\partial a_j^L$, which is elementwise multiplied with the sigmoid derivative. We may also write it in matrix form without using the Hadamard product:

$$\delta^L = \mathbf{S}^L \nabla_a C \tag{18}$$

$$\delta_j^L = \mathrm{diag}(\sigma'(z_j^L))\nabla_a C,$$

where $\mathbf{S}(\mathbf{z}^L)$ is a $j \times j$ diagonal square matrix, whose diagonal entries are the values $\sigma'(z_j^L)$, and the off-diagonal entries are 0. The reason the off-diagonal terms are 0 is because the activation value of neuron j in output layer L does not depend on the weighted input to another neuron in the same layer.

Equation for the error δ^l in terms of the error in the next layer δ^{l+1}:

$$\delta^l = ((\mathbf{W}^{l+1})^T \delta^{l+1}) \odot \sigma'(\mathbf{z}^l), \tag{19}$$

where $(\mathbf{W}^{l+1})^{\mathrm{T}}$ is the transpose of the weight matrix $\mathbf{W}^{l+1}$ for the $(l+1)^{\mathrm{th}}$ layer. Similarly, this equation can be written without the Hadamard product:

$$\delta^l = \mathbf{S}^l (\mathbf{W}^{l+1})^{\mathrm{T}} \delta^{l+1} \tag{20}$$

$$\delta_j^l = \mathrm{diag}(\sigma'(z_j^l))(w_{jk}^{l+1})^{\mathrm{T}} \delta_k^{l+1}$$

Hence, by combining Equations (17) and (19) in iterative fashion, we can derive:

$$\delta^l = \mathbf{S}^l (\mathbf{W}^{l+1})^{\mathrm{T}} \mathbf{S}^{l+1} (\mathbf{W}^{l+2})^{\mathrm{T}} \dots \mathbf{S}^{L-1} (\mathbf{W}^L)^{\mathrm{T}} \mathbf{S}^L \nabla_a C \tag{21}$$

Equation for the rate of change in cost with respect to any bias in the network:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l, \tag{22}$$

that is, the error is δ_j^l is exactly equal to the rate of change $\partial C/\partial b_j^l$.

Equation for the rate of change in cost with respect to any weight in the network:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l, \tag{23}$$

where a_k^{l-1} is the input activation value that comes with the weight w_{jk}^l as input to the j^{th} neuron in the l^{th} layer and δ_j^l is the output error of that neuron. Hence, a weight will learn slowly if either the input neuron is low-activation or if the output neuron has saturated, which means $\sigma'(z_j^l) \approx 0$, giving rise to a small δ_j^l.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(\mathbf{z}^L)$$

$$\delta^l = ((\mathbf{W}^{l+1})^{\mathrm{T}} \delta^{l+1}) \odot \sigma'(\mathbf{z}^l)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \, \delta^l_j$$

We can think of backpropagation as a way of computing the gradient of the cost function by systematically applying the chain rule from multi-variable calculus.

Einstein summation convention will be used in some of the sections below.

The Backpropagation Algorithm:

1. Input $\mathbf{x}$: Provide a training input x_k for each neuron k in the first input layer. The corresponding activation of this input layer is set as $a^1_j = \sigma(w^1_{jk} x_k)$.

$$\mathbf{a}^1 = \sigma(\mathbf{W}^1 \mathbf{x})$$

2. Feedforward: For each layer $l = 2, 3, \ldots, L$ $(l \neq 1)$, compute the weighted inputs $z^l_j = w^l_{jk} a^{l-1}_k + b^l_j$ and the activation values of each neuron in each layer $a^l_j = \sigma(z^l_j)$.

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \quad \text{and} \quad \mathbf{a}^l = \sigma(\mathbf{z}^l)$$

3. Output error δ^L: Compute the error vector of the last output layer $\delta^L_j = (\partial C / \partial a^L_j) \odot (\partial a^L_j / \partial z^L_j)$.

$$\delta^L = \nabla_a C \odot \sigma'(\mathbf{z}^L)$$

4. Backpropagate the error: For each layer $l = L-1, L-2, \ldots, 2$ $(l \neq 1)$, compute the error vectors $\delta^l_j = ((w^{l+1}_{kj})^{\mathrm{T}} \delta^{l+1}_k) \odot \sigma'(z^l_j)$.

$$\delta^l = ((\mathbf{W}^{l+1})^{\mathrm{T}} \delta^{l+1}) \odot \sigma'(\mathbf{z}^l)$$

5. Output gradient: The gradient of the cost function can then be calculated by:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \, \delta^l_j \quad \text{and} \quad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

From inspecting the recipe, the algorithm is called backpropagation because we compute the error vectors δ^l backwards, starting from the final layer. This is a consequence of the fact that the cost is a function of outputs from the network, so we need to start from the final output layer and work backwards through the layers to obtain useful expressions.

Mini-Batch Stochastic Gradient Descent Algorithm with Backpropagation:

1. Input mini-batch m: Provide a mini-batch of m training inputs for the first input layer of the neural network.

2. For each training input $\mathbf{x}$ in the mini-batch m: Set the corresponding input activation $a^{\mathbf{x},1}$, and perform the backpropagation algorithm from Steps 2 to 4:

Feedforward:

$$\mathbf{z}^{\mathbf{x},l} = \mathbf{W}^l \mathbf{a}^{\mathbf{x},l-1} + \mathbf{b}^l \quad \text{and} \quad \mathbf{a}^{\mathbf{x},l} = \sigma(\mathbf{z}^{\mathbf{x},l})$$

Output Error:

$$\delta^{\mathbf{x},L} = \nabla_a C_{\mathbf{x}} \odot \sigma'(\mathbf{z}^{\mathbf{x},L})$$

Backpropagate the Error:

$$\delta^{\mathbf{x},l} = ((\mathbf{W}^{l+1})^{\mathrm{T}} \delta^{\mathbf{x},l+1}) \odot \sigma'(\mathbf{z}^{\mathbf{x},l})$$

3. Gradient Descent: For each layer $l = L, L-1, \ldots, 2$ update the weights and biases for all training inputs in the mini-batch according to the rules:

$$\mathbf{W}^l \to \mathbf{W}^l - \frac{\eta}{m} \sum_{\mathbf{x}} \delta^{\mathbf{x},l} (\mathbf{a}^{\mathbf{x},l-1})^{\mathrm{T}} \tag{24}$$

$$\mathbf{b}^l \to \mathbf{b}^l - \frac{\eta}{m} \sum_{\mathbf{x}} \delta^{\mathbf{x},l} \tag{25}$$

These are simply the rules for updating the variable vector $\mathbf{v} \to \mathbf{v}' = \mathbf{v} - \eta \nabla C$ from Equation (9), where the variable vector includes all weights and biases in the network, and the gradient of the cost function computed using backpropagation. The updates are averaged over a mini-batch m, because the gradient is computed using that mini-batch.

Chapter 3: Improving the Way Neural Networks Learn

By using the sigmoid as an activation function, the neuron learns much more slowly even though it's badly wrong about an outcome. This is quite different from how humans learn, because humans learn fastest when they are very wrong about something.

Inspecting Equations (22) and (23), the gradient of cost depends on the errors δ^l, which in turn depends on the derivative of the sigmoid. The sigmoid derivative becomes very small as it approaches the outputs 0 or 1, which means the cost gradients also become very small. Since the neuron learns by changing the weights and biases at a rate determined by the cost gradient, small cost gradients mean that the neuron learns slowly.

To avoid this slowdown in learning, we use an alternative cost function (not quadratic) called the **cross-entropy cost function**:

$$C = -\frac{1}{n} \sum_x \sum_j y_j \ln a_j^L + (1 - y_j) \ln (1 - a_j^L), \tag{26}$$

where n is the total number of training inputs, the x-sum is over all training inputs x, the j-sum is over all final layer neurons, a_j are the actual output values of final layer neurons and y_j is the corresponding desired output for each final layer neuron j. Similar to the quadratic function, the cost-entropy is positive, and tends toward zero as the neuron gets better at computing the desired output y for all inputs x.

Unlike the quadratic function though, the cost-entropy doesn't slow down because the cost gradients do not have the sigmoid derivative term in it.

The cost-entropy is almost always the better choice, provided the output neurons are sigmoid neurons. However, if the output neurons are linear neurons, i.e. $a_j^L = z_j^L$, then the quadratic cost is an appropriate function to use because there is no slowdown in learning.

We can derive the cross-entropy cost function by integrating the cost gradients obtained from the quadratic cost function $\partial C / \partial w_{jk}^L$ and $\partial C / \partial b_j^L$, with the quadratic gradients divided by the sigmoid derivative $\sigma'(z_j^L)$. In other words, we want a cost function such that there is no sigmoid derivative when we take the gradient.

In [ ]: