Computer Vision

Lecture 06
Multilayer Perceptrons
Military College of Signals

Asim D. Bakhshi
asim.dilawar@mcs.edu.pk

April 29, 2022

Contents

1 Background - No Brain Analogy

It is possible to introduce neural networks without appealing to brain analogies. In the leacture on linear classification we computed scores for different visual categories given the image using the scoring formula $s = \mathbf{W}x$, where $\mathbf{W}$ was a matrix and x was an input column vector containing all pixel data of the image. In the case of CIFAR-10, x is a $[3072 \times 1]$ column vector, and $\mathbf{W}$ is a $[10 \times 3072]$ matrix, so that the output scores is a vector of 10 class scores.

An example neural network would instead compute $s = \mathbf{W}_2 \max(0, \mathbf{W}_1 x)$. Here, $\mathbf{W}_1$ could be, for example, a $[10 \times 3072]$ matrix transforming the image into a 100-dimensional intermediate vector. The function $\max(0, -)$ is a non-linearity that is applied elementwise. There are several choices we could make for the non-linearity (which we'll study below), but this one is a common choice and simply thresholds all activations that are below zero to zero. Finally, the matrix $\mathbf{W}2$ would then be of size $[10 \times 100]$, so that we again get 10 numbers out that we interpret as the class scores. Notice that the non-linearity is critical computationally - if we left it out, the two matrices could be collapsed to a single matrix, and therefore the predicted class scores would again be a linear function of the input. The non-linearity is where we get the wiggle. The parameters $\mathbf{W}_2, \mathbf{W}_1$ are learned with stochastic gradient descent, and their gradients are derived with chain rule (and computed with backpropagation).

A three-layer neural network could analogously look like

$$s = \mathbf{W}_3 \max(0, \mathbf{W}_2 \max(0, \mathbf{W}_1 x))$$

where all of $\mathbf{W}_3, \mathbf{W}_2, \mathbf{W}_1$ are parameters to be learned. The sizes of the intermediate hidden vectors are hyperparameters of the network and we'll see how we can set them later. Lets now look into how we can interpret these computations from the neuron/network perspective.

2 Modelling a Single Neuron

The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems, but has since diverged and become a

matter of engineering and achieving good results in Machine Learning tasks. Nonetheless, we begin our discussion with a very brief and high-level description of the biological system that a large portion of this area has been inspired by.

2.1 Biological Motivation

The basic computational unit of the brain is a neuron. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^{14} to 10^{15} synapses. Figure 1 shows a cartoon drawing of a biological neuron (left) and a common mathematical model (right). Each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model of a neuron, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. $w_0 x_0$) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory (positive weight) or inhibitory (negative weight)) of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can fire, sending a spike along its axon. In the computational model, we assume that the precise timings of the spikes do not matter, and that only the frequency of the firing communicates information. Based on this rate code interpretation, we model the firing rate of the neuron with an **activation function** f, which represents the frequency of the spikes along the axon. Historically, a common choice of activation function is the **sigmoid function** σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0 and 1. We will see details of these activation functions later in this section.

In the mathematical context, each neuron performs a dot product with the input and its weights, adds the bias and applies the non-linearity (or activation function), in this case the sigmoid

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

We will go into more details about different activation functions at the end of this section.

Coarse model . It's important to stress that this model of a biological neuron is very coarse: For example, there are many different types of neurons, each with different properties. The dendrites in biological neurons perform complex nonlinear computations. The synapses are not just a single weight, they're a complex non-linear dynamical system. The exact timing of the output spikes in many systems is known to be important, suggesting that the rate code approximation may not hold. Due to all these and many other simplifications, be

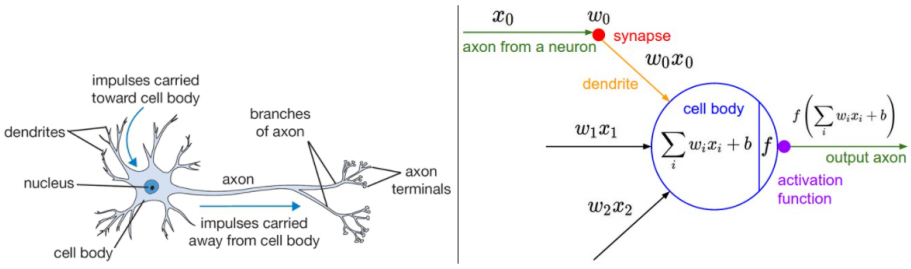

Figure 1: Drawing of a biological neuron (left) and its mathematical model (right).

prepared to hear groaning sounds from anyone with some neuroscience background if you draw analogies between Neural Networks and real brains. See this review [1], or more recently this review [2] if you are interested.

2.2 Single Neuron as a Linear Classifier

The mathematical form of the model Neuron's forward computation might look familiar to you. As we saw with linear classifiers, a neuron has the capacity to *like* (activation near one) or *dislike* (activation near zero) certain linear regions of its input space. Hence, with an appropriate loss function on the neuron's output, we can turn a single neuron into a linear classifier:

2.2.1 Binary Softmax Classifier

For example, we can interpret $\sigma(\sum_i w_i x_i + b)$ to be the probability of one of the classes $P(y_i = 1 \mid x_i; w)$. The probability of the other class would be $P(y_i = 0 \mid x_i; w) = 1 - P(y_i = 1 \mid x_i; w)$, since they must sum to one. With this interpretation, we can formulate the cross-entropy loss as we have seen in the Linear Classification lecture, and optimizing it would lead to a binary Softmax classifier (also known as logistic regression). Since the sigmoid function is restricted to be between $0 - 1$, the predictions of this classifier are based on whether the output of the neuron is greater than 0.5.

2.2.2 Binary SVM Classifier

Alternatively, we could attach a max-margin hinge loss to the output of the neuron and train it to become a binary Support Vector Machine.

2.2.3 Regularization Interpretation

The regularization loss in both SVM/Softmax cases could in this biological view be interpreted as gradual forgetting, since it would have the effect of driving all synaptic weights w towards zero after every parameter update.

2.3 Commonly Used Activation Function

Every activation function (or non-linearity) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions you may encounter in practice.

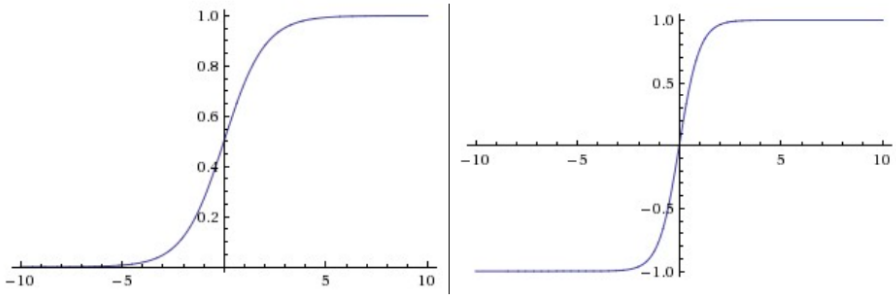

Figure 2: Left: Sigmoid non-linearity squashes real numbers to range between $[0, 1]$ Right: The tanh non-linearity squashes real numbers to range between $[-1, 1]$

2.3.1 Sigmoid

The sigmoid non-linearity has the mathematical form

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

and is shown in Figure 2 on the left. As alluded to previously, it takes a real-valued number and "squashes" it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:

- Sigmoids saturate and kill gradients. A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights

are too large then most neurons would become saturated and the network will barely learn.

- Sigmoid outputs are not zero-centered. This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = w^T x + b$)), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

2.3.2 Tanh

The tanh non-linearity is shown on the image above on the right. It squashes a real-valued number to the range $[-1, 1]$. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds:

$$\tanh(x) = 2\sigma(2x) - 1$$

2.3.3 ReLU

The Rectified Linear Unit has become very popular in the last few years. It computes the function $f(x) = \max(0, x)$. In other words, the activation is simply thresholded at zero (see Figure 3 on the left). There are several pros and cons to using the ReLUs:

- (+) It was found to greatly accelerate (e.g. a factor of 6 in [3].) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- (−) Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will

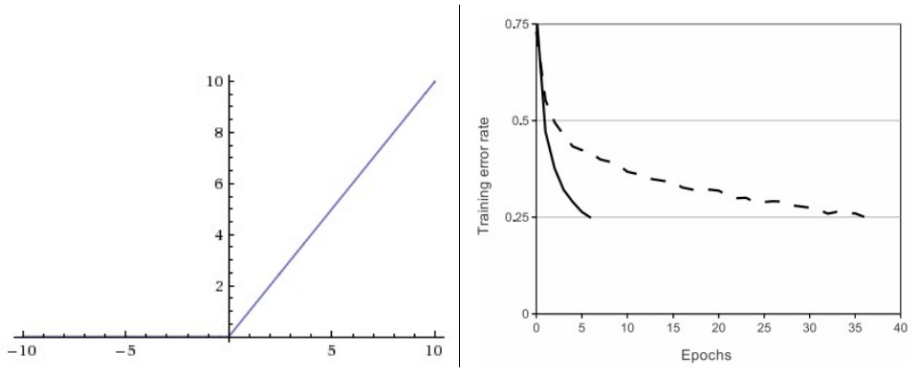

Figure 3: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$. Right: A plot from [3] paper indicating the 6× improvement in convergence with the ReLU unit compared to the tanh unit.

never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

2.3.4 Leaky ReLU

Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small positive slope (of 0.01, or so). That is, the function computes

$$f(x) = \mathbf{1}(x < 0)(\alpha x) + \mathbf{1}(x >= 0)(x)$$

where σ is a small constant. Some people report success with this form of activation function, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons, introduced in [4]. However, the consistency of the benefit across tasks is presently unclear.

2.3.5 Maxout

Other types of units have been proposed that do not have the functional form $f(w^T x + b)$ where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the Maxout neuron

(introduced recently by [5].) that generalizes the ReLU and its leaky version. The Maxout neuron computes the function

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $w_1, b_1 = 0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

This concludes our discussion of the most common types of neurons and their activation functions. As a last comment, it is very rare to mix and match different types of neurons in the same network, even though there is no fundamental problem with doing so.

TLDR : What neuron type should I use? Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of "dead" units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

3 Neural Network Architectures

3.1 Layer-wise organization

3.1.1 Fully-Connected Layer

Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the fully-connected layer in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. Figure 4 shows two example Neural Network topologies that use a stack of fully-connected layers.

3.1.2 Naming Conventions

Notice that when we say N-layer neural network, we do not count the input layer. Therefore, a single-layer neural network describes a network with no hidden layers (input directly mapped to output). In that sense, you can sometimes hear people say that logistic regression or SVMs are simply a special case of single-layer Neural Networks. You may also hear these networks interchangeably

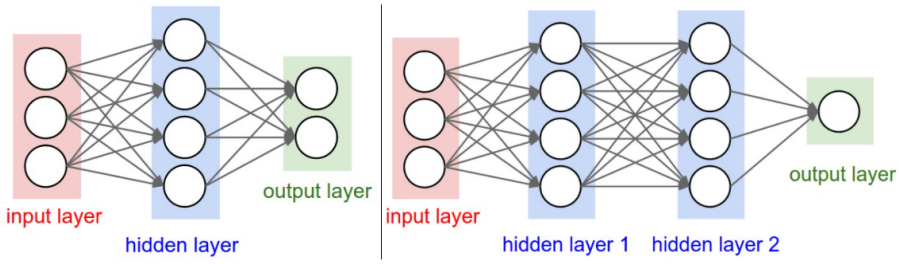

Figure 4: Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

referred to as Artificial Neural Networks (ANN) or Multi-Layer Perceptrons (MLP). Many people do not like the analogies between Neural Networks and real brains and prefer to refer to neurons as units.

In case of MLPs each layer contains some number of identical units. Every unit in one layer is connected to every unit in the next layer; we say that the network is fully connected. The first layer is the **input layer**, and its units take the values of the input features. The last layer is the **output layer**, and it has one unit for each value the network outputs (i.e. a single unit in the case of regression or binary classifiation, or K units in the case of K-class classification).

All the layers in between these are known as **hidden layers**, because we don't know ahead of time what these units should compute, and this needs to be discovered during learning. The units in these layers are known as **input units, output units, and hidden units**, respectively. The number of layers is known as the **depth**, and the number of units in a layer is known as the **width**. Terminology for the depth is very inconsistent. A network with one hidden layer could be called a one-layer, two-layer, or three-layer network, depending if you count the input and output layers. As you might guess, **deep learning** refers to training neural nets with many layers.

3.1.3 Output Layer

Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).

3.1.4 Sizing Neural Networks

The two metrics that people commonly use to measure the size of neural networks are the number of neurons, or more commonly the number of parameters. Working with the two example networks in Figure 4:

- The first network (left) has $4 + 2 = 6$ neurons (not counting the inputs), $[3 \times 4] + [4 \times 2] = 20$ weights and $4 + 2 = 6$ biases, for a total of 26 learnable parameters.
- The second network (right) has $4+4+1 = 9$ neurons, $[3x4]+[4\times4]+[4\times1] = 12+16+4 = 32$ weights and $4+4+1 = 9$ biases, for a total of 41 learnable parameters.

To give you some context, modern Convolutional Networks contain on orders of 100 million parameters and are usually made up of approximately 10-20 layers (hence deep learning). However, as we will see the number of effective connections is significantly greater due to parameter sharing.

3.2 Multiplayer Perceptrons

We can describe a general neuron-like processing unit:

$$y = f\left(\sum_j w_j x_j + b\right) \tag{1}$$

where the x_j are the inputs to the unit, the w_j are the weights, b is the bias, $f(\cdot)$ is the nonlinear activation function, and y is the unit's activation. We have seen a bunch of examples of such units as follows:

- **Linear regression:** uses a linear model, so $f(z) = z$.
- **Binary linear classifiers:** $f(\cdot)$ is a hard threshold at zero.
- **Logistic regression**: $f(z)$ is the logistic function $f(z) = \frac{1}{(1+e^{-z})}$.

A neural network is just a combination of lots of these units. Each one performs a very simple and stereotyped function, but in aggregate they can do some very useful computations. For now, we we will concern ourselves with feed-forward neural networks, where the units are arranged into a graph without any cycles, so that all the computation can be done sequentially. This is in contrast with recurrent neural networks, where the graph can have cycles, so the processing can feed into itself. These are much more complicated, and we will cover them later.

We have already mentioned that a simplest kind of feed-forward network is a multilayer perceptron (MLP).

3.3 Example: XOR Function

As an example to illustrate the power of MLPs, let us design one that computes the XOR function. Remember, we showed that linear models cannot do this. We can verbally describe XOR as **one of the inputs is 1, but not both of them.** So let us keep a hidden unit h_1 to detect if at least one of the inputs is 1, and have h_2 detect if they are both 1.

We can easily do this if we use a hard threshold activation function. You already know how to design such units since it is just an exercise of designing a binary linear classifier. Then the output unit will activate only if $h_1 = 1$ and $h_2 = 0$.

Now let us write out the MLP computations mathematically. Conceptually, there is nothing new here as well; we just have to pick a notation to refer to various parts of the network. As with the linear case, we will refer to the activations of the input units as x_j and the activation of the output unit as y. The units in the lth hidden layer will be denoted $h_i^{(l)}$.

Our network is **fully connected**, so each unit receives connections from all the units in the previous layer. This means each unit has its own bias, and there is a weight for every pair of units in two consecutive layers. Therefore, the network computations can be written out as:

$$\begin{aligned} h_i^{(1)} &= f^{(1)}\left(\sum_j w_{ij}^{(1)} x_j + b_i^{(1)}\right) \\ h_i^{(2)} &= f^{(2)}\left(\sum_j w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)}\right) \\ y_i &= f^{(3)}\left(\sum_j w_{ij}^{(3)} h_j^{(2)} + b_i^{(3)}\right) \end{aligned} \tag{2}$$

Note that we distinguish $f^{(1)}$ and $f^{(2)}$ because different layers may have different activation functions.

Since all these summations and indices can be cumbersome, we usually write the computations in vectorized form. Since each layer contains multiple units, we represent the activations of all its units with an activation vector $\mathbf{h}^{(l)}$. Since there is a weight for every pair of units in two consecutive layers, we represent each layer's weights with a weight matrix $\mathbf{W}^{(l)}$. Each layer also has a bias vector $\mathbf{b}^{(l)}$. The above computations are therefore written in vectorized form as:

$$\begin{aligned} \mathbf{h}^{(1)} &= f^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \\ \mathbf{h}^{(2)} &= f^{(2)}\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right) \\ \mathbf{y}_i &= f^{(3)}\left(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}\right) \end{aligned} \tag{3}$$

When we write the activation function applied to a vector, this means it is applied independently to all the entries. Recall how in linear regression, we combined all the training examples into a single matrix $\mathbf{X}$, so that we could compute all the predictions using a single matrix multiplication. We can do the same thing here. We can store all of each layer's hidden units for all the training examples as a matrix $\mathbf{H}^{(l)}$. Each row contains the hidden units for one example. The computations are written as follows (note the transposes):

$$\begin{aligned} \mathbf{H}^{(1)} &= f^{(1)}\left(\mathbf{X}\mathbf{W}^{(1)^\top} + \mathbf{1}\mathbf{b}^{(1)^\top}\right) \\ \mathbf{H}^{(2)} &= f^{(2)}\left(\mathbf{H}^{(1)}\mathbf{W}^{(2)^\top} + \mathbf{1}\mathbf{b}^{(2)^\top}\right) \\ \mathbf{Y} &= f^{(3)}\left(\mathbf{H}^{(2)}\mathbf{W}^{(3)^\top} + \mathbf{1}\mathbf{b}^{(3)^\top}\right) \end{aligned} \tag{4}$$

These equations can be translated directly into NumPy code which efficiently computes the predictions over the whole dataset.

3.4 Context of Feature Learning

Neural nets can be thought of as a way of learning nonlinear feature mappings. The last hidden layer can be thought of as a feature map, and the output layer weights can be thought of as a linear model using those features. But the whole thing can be trained end-to-end with **backpropagation**, which we have covered in the last lecture. The hope is that we can learn a feature representation where the data become linearly separable.

3.5 Backpropagation

Computing the loss derivatives of a multilayer neural network is just a prototypical case of applying backpropagation algorithm. We can revisit the general configuration of an MLP as

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \tag{5}$$

$$h_i = f(z_i) \tag{6}$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \tag{7}$$

$$\mathcal{L} = \frac{1}{2}\sum_k (y_k - t_k)^2 \tag{8}$$

As before, we start by drawing out the computation graph for the network. The case of two input dimensions and two hidden units is shown in Figure 5(a). Because the graph clearly gets pretty cluttered if we include all the units individually, we can instead draw the computation graph for the vectorized form (Figure 5(b)), as long as we can mentally convert it to Figure 5(a) as needed.

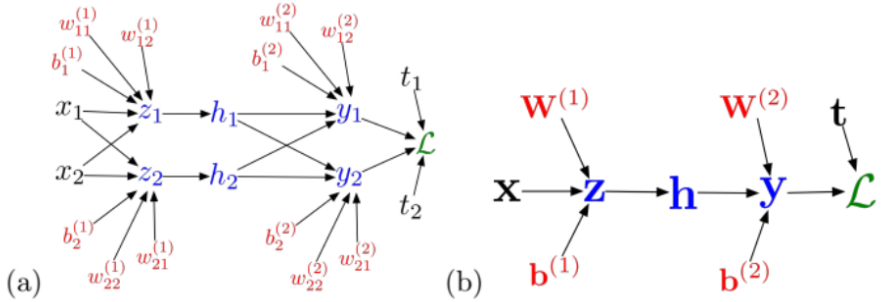

Figure 5: (a) Full computation graph for the loss computation in a multi-layer neural net. (b) Vectorized form of the computation graph

Based on this computation graph, we can work through the derivations of the backwards pass just as before.

$$
\begin{aligned}
\frac{\partial \mathcal{E}}{\partial \mathcal{L}} &= 1 \\
\frac{\partial \mathcal{E}}{\partial y_k} &= \frac{\partial \mathcal{E}}{\partial \mathcal{L}} \cdot (y_k - t_k) \\
\frac{\partial \mathcal{E}}{\partial w_{ki}^{(2)}} &= \frac{\partial \mathcal{E}}{\partial y_k} \cdot h_i \\
\frac{\partial \mathcal{E}}{\partial b_k^{(2)}} &= \frac{\partial \mathcal{E}}{\partial y_k} \\
\frac{\partial \mathcal{E}}{\partial h_i} &= \sum_k \frac{\partial \mathcal{E}}{\partial y_k} \cdot w_{ki}^{(2)} \\
\frac{\partial \mathcal{E}}{\partial z_i} &= \frac{\partial \mathcal{E}}{\partial h_i} \cdot f'(z_i) \\
\frac{\partial \mathcal{E}}{\partial w_{ij}^{(1)}} &= \frac{\partial \mathcal{E}}{\partial z_i} \cdot x_j \\
\frac{\partial \mathcal{E}}{\partial b_i^{(1)}} &= \frac{\partial \mathcal{E}}{\partial z_i}
\end{aligned}
$$

Once we have derived the update rules in terms of indices, we can find the vectorized versions the same way we have been doing for all our other calculations. For the forward pass:

$$
\begin{aligned}
\mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{h} &= \sigma(\mathbf{z}) \\
\mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\
\mathcal{L} &= \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2
\end{aligned}
$$

And the backward pass:

$$
\begin{aligned}
\frac{\partial \mathcal{E}}{\partial \mathcal{L}} &= 1 \\
\frac{\partial \mathcal{E}}{\partial \mathbf{y}} &= \frac{\partial \mathcal{E}}{\partial \mathcal{L}} \cdot (\mathbf{y} - \mathbf{t}) \\
\frac{\partial \mathcal{E}}{\partial \mathbf{W}^{(2)}} &= \frac{\partial \mathcal{E}}{\partial \mathbf{y}} \cdot \mathbf{h}^\top \\
\frac{\partial \mathcal{E}}{\partial \mathbf{b}^{(2)}} &= \frac{\partial \mathcal{E}}{\partial \mathbf{y}} \cdot 1 \\
\frac{\partial \mathcal{E}}{\partial \mathbf{h}} &= \mathbf{W}^{(2)^\top} \cdot \frac{\partial \mathcal{E}}{\partial \mathbf{y}} \\
\frac{\partial \mathcal{E}}{\partial \mathbf{z}} &= \frac{\partial \mathcal{E}}{\partial \mathbf{h}} \cdot f'(\mathbf{z}) \\
\frac{\partial \mathcal{E}}{\partial \mathbf{W}^{(1)}} &= \frac{\partial \mathcal{E}}{\partial \mathbf{z}} \cdot \mathbf{x}^\top \\
\frac{\partial \mathcal{E}}{\partial \mathbf{b}^{(1)}} &= \frac{\partial \mathcal{E}}{\partial \mathbf{z}} \cdot 1
\end{aligned}
$$

3.6 Example: MNIST Handwritten Digits

Consider training an MLP to recognize handwritten digits. The input is a 28×28 grayscale image, and all the pixels take values between 0 and 1. We'll ignore the spatial structure, and treat each input as a $\mathbb{R}^{784}$ vector. Later on, we'll talk about convolutional networks, which use the spatial structure of the image. This is a multiway classification task with 10 categories, one for each digit class. Suppose we train an MLP with two hidden layers. We can try to understand what the first layer of hidden units is computing by visualizing the weights. Each hidden unit receives inputs from each of the pixels, which means the weights feeding into each hidden unit can be represented as an $\mathbb{R}^{784}$ vector, the same as the input size.

If we visualize the layers, positive values will be lighter, and negative values will be darker. Each hidden unit computes the dot product of these vectors with the input image, and then passes the result through the activation function. So if the light regions of the filter overlap the light regions of the image, and the dark regions of the filter overlap the dark region of the image, then the unit will activate. We'll see that oriented edges are a very commonly learned by the

first layers of neural nets for visual processing tasks. It's harder to visualize what the second layer is doing. There are indeed tricks for visualizing this. In a nutshell, higher layers of a neural net can learn increasingly high-level and complex features.

References

[1] Michael London and Michael Häusser. "Dendritic computation". *Annu. Rev. Neurosci.* 28, pp. 503–532, 2005.

[2] Nicolas Brunel, Vincent Hakim, and Magnus JE Richardson. "Single neuron dynamics and computation". *Current opinion in neurobiology* 25, pp. 149–155, 2014.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". *Advances in neural information processing systems* 25, pp. 1097–1105, 2012.

[4] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015. Pp. 1026–1034.

[5] Ian Goodfellow et al. "Maxout networks". In: *International conference on machine learning*. PMLR. 2013. Pp. 1319–1327.