

6.036 Spring 2018 : Week 4

February 25, 2018

5 Learning representations – Neural networks

- feed-forward computation in neural networks
- hidden layers as feature maps for classification
- back-propagation for evaluating gradients

We have have already covered on way of performing non-linear classification. The idea is to map examples x explicitly into feature vectors $\phi(x)$ which contain non-linear terms of the coordinates of x. The classification decisions are based on

$$\hat{y} = \text{sign}(\theta \cdot \phi(x)) \tag{1}$$

where we have omitted the bias/offset term for simplicity. However, this mapping is not learned specifically to improve classification performance.

We will next formulate models – neural networks – where the feature representation is learned jointly with the classifier, both focused on improving the end-to-end performance.

5.1 Feed-forward Neural Networks

Neural networks consist of a large number of simple computational units/neurons (e.g., linear classifiers) which, together, specify how the input vector x is processed towards the final classification decision. Neural networks can do much more than just classification but we will use classification problems here for continuity. In a simple case, the units in the neural network are arranged in layers, where each layer defines how the input signal is transformed in stages. These are called *feed-forward* neural networks. They are loosely motivated by how our visual system processes the signal coming to the eyes in massively parallel stages. The layers in our models include

1. *input layer* where the units simply store the coordinates of the input vector (one unit assigned to each coordinate). The input units are special in the sense that they don't compute anything. Their activation is just the value of the corresponding input coordinate.

2. possible *hidden layers* of units which represent complex transforms of the input signal, from one layer to the next, towards the final classifier. These units determine their activation by aggregating input from the preceding layer

3. *output layer*, here a single unit, which is a linear classifier taking as its input the activations of the units in the penultimate (hidden or the input) layer.

Figure 1 shows a simple feed-forward neural network with two hidden layers. The units correspond to the nodes in the graph and the edges specify how the activation of one unit may depend on the activation of other units. More specifically, each unit aggregates input from other preceding units (units in the previous layer) and evaluates an output/activation value by passing the summed input through a (typically non-linear) transfer/activation/link function. All the units except the input units act in this way. The input units are just clamped to the observed coordinates of x.

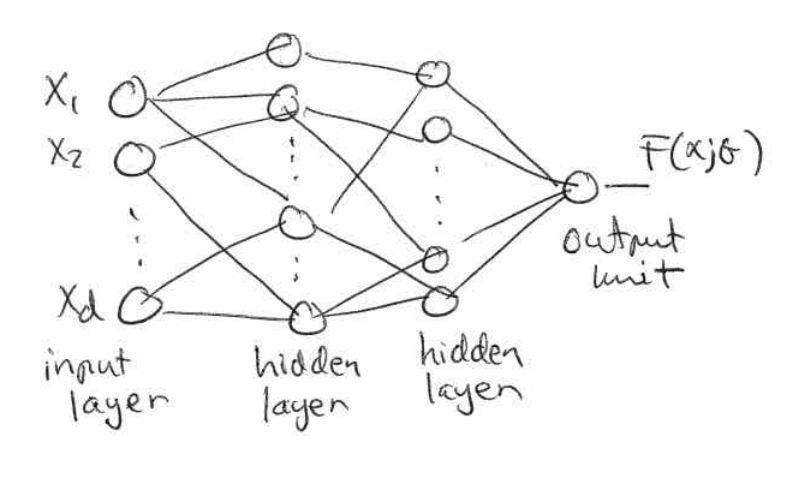

Figure 1: A feed-forward neural network with two hidden layers and a single output unit.

5.2 Simplest neural network

Let's begin with the simplest neural network (a linear classifier). In this case, we only have d input units corresponding to the coordinates of $x = [x_1, \ldots, x_d]^T$ and a single linear output unit producing $F(x; \theta)$ shown in Figure 2. We will use θ to refer to the set of all parameters in a given network. So the number of parameters in θ varies from one architecture to another. Now, the output unit receives as its aggregated input a weighted combination of the input units (plus an overall offset w_0).

$$z = \sum_{i=1}^{d} x_i w_i + w_0 = x \cdot w + w_0 \quad \text{(weighted summed input to the unit)} \tag{2}$$

$$F(x; \theta) = f(z) = z \quad \text{(network output)} \tag{3}$$

where the activation function $f(\cdot)$ is simply linear $f(z) = z$. So this is just a standard linear classifier if we classify each x by taking the sign of the network output. We will leave the network output as a real number, however, so that it can be easily fed into the Hinge or

other such loss function. The parameters θ in this case are $\theta = \{w_1, \ldots, w_d, w_0\} = \{w, w_0\}$ where w is a vector $w = [w_1, \ldots, w_d]^T$.

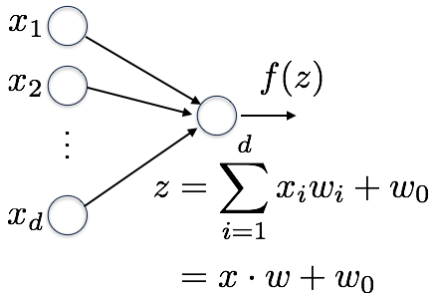

Figure 2: A simple neural network with no hidden units.

Note that we can interpret and draw each unit in a neural network graphically as a linear classifier based on the inputs that it receives. We draw a "decision boundary" corresponding to all x such that the aggregate input to the unit is zero or $\{x : x \cdot w + w_0 = 0\}$. The direction in $x-$space in which the aggregate input z increases is indicated by the parameter vector w and how fast it increases away from the all-zero boundary is exactly $\|w\|$. This is exactly as in a linear classifier. The output of the unit may be non-linear in general, evaluated as $f(z)$ where $f(\cdot)$ is typically a monotonically increasing (non-decreasing) function of z (e.g., linear as above). The graphical interpretation as a linear classifier is therefore useful even with the non-linearity.

5.3 Hidden layers, computation

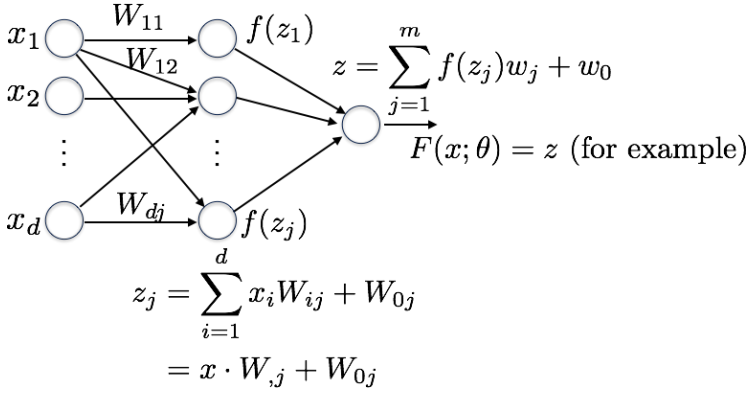

Figure 3: A neural network with d input units, m non-linear hidden units, and one linear output unit.

Let's extend the model a bit and consider a neural network with one hidden layer. This is shown in Figure 3. As before, the input units are simply clamped to the coordinates of the

input vector x. In contrast, each of the m hidden units evaluate their output in two steps

$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j} \text{ (weighted input to the } j^{th} \text{ hidden unit)} \quad (4)$$

$$f(z_j) = \max\{0, z_j\} \text{ (rectified linear activation function)} \quad (5)$$

where we have used so called *rectified linear* activation function which simply passes any positive input through as is and squashes any negative input to zero. A number of other activation functions are possible, including $f(z) = \text{sign}(z)$, $f(z) = (1 + \exp(-z))^{-1}$ (sigmoid), $f(z) = \tanh(z)$, and so on. We will use rectified linear hidden units for illustrations as they are convenient when we derive the learning algorithm. Now, the single output unit no longer sees the input example directly but only through the activations of the hidden units. In other words, as a unit, it can be written exactly as before but it now takes in each $f(z_j)$ as input instead of the coordinates x_i. Thus

$$z = \sum_{j=1}^{m} f(z_j) w_j + w_0 \text{ (weighted summed input to the unit)} \quad (6)$$

$$F(x; \theta) = z \text{ (network output)} \quad (7)$$

The output unit is again linear and functions as a linear classifier on a new feature representation $[f(z_1), \ldots, f(z_m)]^T$ of each example. Note that z_j are always functions of x but we have suppressed this dependence in the notation. The parameters θ in this case include both the weights $\{W_{,j}, W_{0j}\}$, where $W_{,j}$ is a vector for each j, that mediate hidden unit activations in response to the input, and $\{w, w_0\}$, where w is a vector, which specify how the network output depends on the hidden units.

5.4 Hidden layers, representation

How powerful is the neural network model with one hidden layer? It turns out that it is already a universal approximator in the sense that it can approximate any mapping from the input to the output if we increase the number of hidden units. But it is not necessarily easy to find the parameters θ that realize any specific mapping exemplified by the training examples. We will return to this question later.

Let's take a simple example to see where the power lies. Consider the labeled two dimensional points shown in Figure 4 (left). The points are clearly not linearly separable and therefore cannot be correctly classified with a simple neural network without hidden units. Suppose we introduce only two hidden units such that

$$z_1 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} W_{11} \\ W_{21} \end{bmatrix} + W_{01} \quad (8)$$

$$f(z_1) = \max\{0, z_1\} \text{ (activation of the 1st hidden unit)} \quad (9)$$

$$z_2 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} W_{12} \\ W_{22} \end{bmatrix} + W_{02} \quad (10)$$

$$f(z_2) = \max\{0, z_2\} \text{ (activation of the 2nd hidden unit)} \quad (11)$$

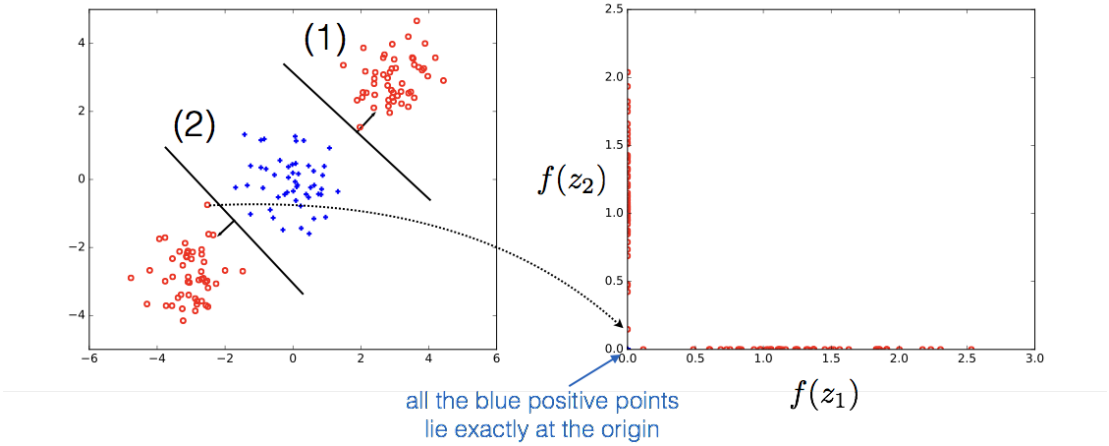

Figure 4: Hidden unit activations and examples represented in hidden unit coordinates

Each example x is then first mapped to the activations of the hidden units, i.e., has a two-dimensional feature representation $[f(z_1), f(z_2)]^T$. We choose the parameters W as shown pictorially in the Figure 4 (left). Note again that we can represent the hidden unit activations similarly to a decision boundary in a linear classifier. The only difference is that the output is not binary but rather is identically zero in the negative half, and increases linearly in the positive part as we move away from the boundary. Now, using these parameters, we can map the labeled points to their feature coordinates $[f(z_1), f(z_2)]^T$ as shown on the right in the same figure. The labeled points are now linearly separable in these feature coordinates and therefore the single output unit – a linear classifier – can find a separator that correctly classifies these training examples.

As an exercise, think about whether the examples remain linearly separable in the feature coordinates $[f(z_1), f(z_2)]^T$ if we flip the positive/negative sides of the two hidden units. In other words, now the positive examples would have $f(z_1) > 0$ and $f(z_2) > 0$. This example highlights why parameter estimation in neural networks is not an easy task.

5.5 Learning, stochastic gradient

Given a training set $\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$ of examples $x \in \mathbb{R}^d$ and labels $y \in \{-1, 1\}$, we would like to estimate parameters θ of the chosen neural network model so as to minimize the average loss over the training examples,

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) \tag{12}$$

where we assume that the loss is the Hinge loss $\text{Loss}(z) = \max\{0, 1 - z\}$. Clearly, since these models can be very complex, we should add a regularization term as well. We could use, for example, $\|\theta\|^2/2$ as the regularizer so as to squash parameters towards zero if they are not helpful for classification. There is a better way to regularize neural network models so we will leave the regularization out for now, and return to this question later.

In order to minimize the average loss, we will resort to a simple stochastic optimization procedure rather than performing gradient descent steps on $J(\theta)$ directly. The stochastic version, while simpler, is also likely to work better with complex models, providing the means to randomize the exploration of good parameter values. On a high level, our algorithm is simply as follows. We sample a training example at random, and nudge the parameters towards values that would improve the classification of that example. Many such small steps will overall move the parameters in a direction that reduce the average loss. The algorithm is known as *stochastic gradient descent* or SGD, written more formally as

$$\text{(0) Initialize } \theta \text{ to small random values} \tag{13}$$

$$\text{(1) Select } i \in \{1, \ldots, n\} \text{ at random or in a random order} \tag{14}$$

$$\text{(2) } \theta \leftarrow \theta - \eta_k \nabla_\theta \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) \tag{15}$$

where we iterate between (1) and (2). Here the gradient

$$\nabla_\theta \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) = [\frac{\partial}{\partial \theta_1} \text{Loss}(y^{(i)} F(x^{(i)}; \theta)), \ldots, \frac{\partial}{\partial \theta_D} \text{Loss}(y^{(i)} F(x^{(i)}; \theta))]^T \tag{16}$$

has the same dimension as the parameter vector, and it points in a direction in the parameter space where the loss function increases. We therefore nudge the parameters in the opposite direction. The *learning rate* η_k should decrease slowly with the number of updates. It should be small enough that we don't overshoot (often), i.e., if η_k is large, the new parameter values might actually increase the loss after the update. If we set, $\eta_k = \eta_0/(k+1)$ or, more generally, set η_k, $k = 1, 2, \ldots$, such that $\sum_{k=1}^{\infty} \eta_k = \infty$, $\sum_{k=1}^{\infty} \eta_k^2 < \infty$, then we would be guaranteed in simple cases (without hidden layers) that the algorithm converges to the minimum average loss. The presence of hidden layers makes the problem considerably more challenging. For example, can you see why it is critical that the parameters are NOT initialized to zero when we have hidden layers? We will make the algorithm more concrete later, actually demonstrating how the gradient can be evaluated by propagating the training signal from the output (where we can measure the discrepancy) back towards the input layer (where many of the parameters are).

While our estimation problem may appear daunting with lots of hidden units, it is surprisingly easier if we increase the number of hidden units in each layer. In contrast, adding layers (deeper architectures) are tougher to optimize. The argument for increasing the size of each layer is that high-dimensional intermediate feature spaces yield much room for gradient steps to traverse while permitting continued progress towards minimizing the objective. This is not the case with small models. To illustrate this effect, consider the classification problem in Figure 5. Note that the points are setup similarly to the example discussed above where only two hidden units would suffice to solve the problem. We will try networks with a single hidden layer, and 10, 100, and 500 hidden units. It may seem that the smallest of them – 10 hidden units – should already easily solve the problem. While it clearly has the power to do so, such parameters are hard to find with gradient based algorithms (repeated attempts with random initialization tend to fail as in the figure). However, with 100 or 500 hidden units, we can easily find a nice decision boundary that separates the examples.

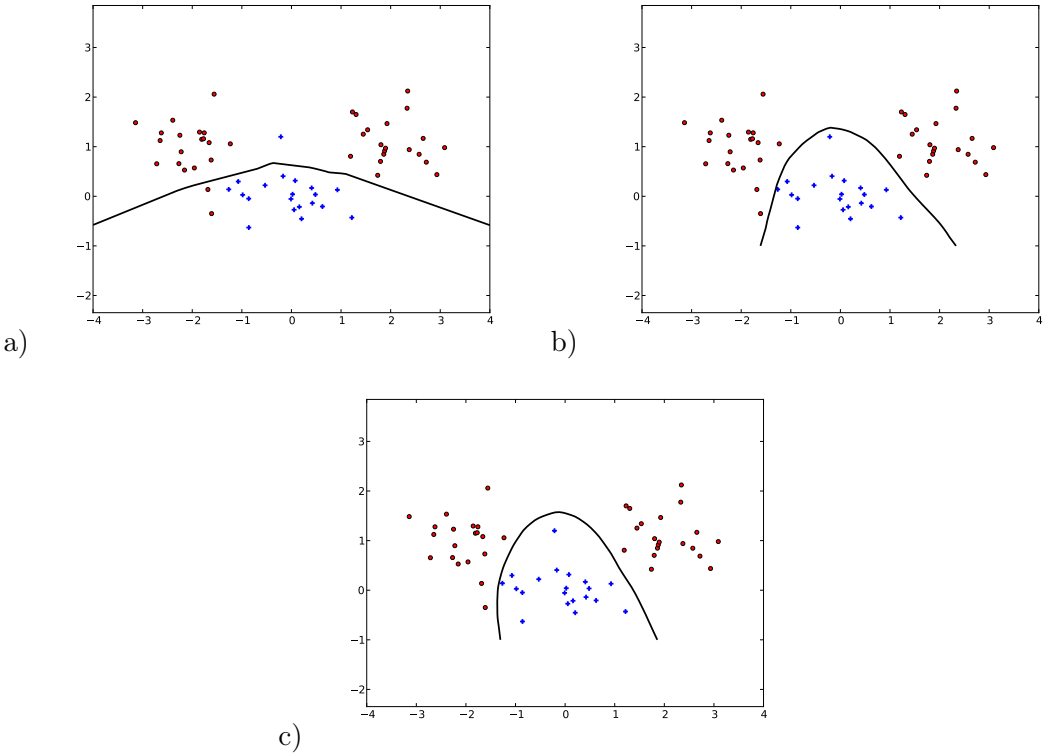

Figure 5: Decision boundaries resulting from training neural network models with one hidden layer and varying number of hidden units: a) 10; b) 100; and c) 500.

5.5.1 Back-propagation and SGD

Our goal here is to make the algorithm more concrete, adapting it to simple example networks, and seeing back-propagation (chain rule) in action. We will also discuss little adjustments to the algorithm that may help make it work better in practice.

If viewed as a classifier, our network generates a real valued output $F(x;\theta)$ in response to any input vector $x \in \mathbb{R}^d$. This mapping is mediated by parameters θ that represent all the weights in the model. For example, in case of a) no hidden units, i.e., a linear classifier, or b) two layer neural network, the parameters θ correspond to vectors

$$\theta = [w_1, \ldots, w_d, w_0]^T \quad (17)$$

$$\theta = [W_{11}, \ldots, W_{1m}, \ldots, W_{d1}, \ldots, W_{dm}, W_{01}, \ldots, W_{0m}, w_1, \ldots, w_d, w_0]^T \quad (18)$$

respectively. You can compare these to Figures 2 and 3 above.

We will illustrate here first how to use stochastic gradient descent (SDG) to minimize average loss over the training examples shown in Eq.(12). The algorithm performs a series of small updates by focusing each time on a randomly chosen loss term. After many such small updates, the parameters will have moved in a direction that reduces the overall loss above. More concretely, we iteratively select a training example $(x^{(t)}, y^{(t)})$ at random (or in

a random order) and move the parameters in the opposite direction of the gradient of the loss associated with that example. Each parameter update is therefore of the form

$$\theta \leftarrow \theta - \eta_k \nabla_\theta \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big) \tag{19}$$

where η_k is the learning rate/step-size parameter after k updates. Note that we will update all the parameters in each step. In other words, if we break the update into each coordinate, then

$$\theta_i \leftarrow \theta_i - \eta_k \frac{\partial}{\partial \theta_i} \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big), \quad i = 1, \ldots, D \tag{20}$$

where D is the total number of parameters in the network. In order to use SGD, we need to specify three things: 1) how to evaluate the derivatives, 2) how to initialize the parameters, and 3) how to set the learning rate.

Simple case: no hidden units

Let's begin with the simplest network without hidden units as in Figure 2 where $\theta = [w_1, \ldots, w_d, w_0]^T$. This is just a linear classifier and thus SGD can be obtained as with the Pegasos algorithm. We will go through the calculation in detail since we will structure it in a manner that generalizes to more complex models. Now, given any training example $(x^{(t)}, y^{(t)})$, our first task is to evaluate

$$\frac{\partial}{\partial w_i} \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big) \quad = \quad \frac{\partial}{\partial w_i} \text{Loss}\big(y^{(t)} z^{(t)}\big) \tag{21}$$

where $z^{(t)} = \sum_{j=1}^d x_j^{(t)} w_j + w_0$ and we have assumed (as before) that the output unit is linear. Moreover, we will assume that the loss function is the Hinge loss, i.e., $\text{Loss}(yz) = \max\{0, 1 - yz\}$. Now, the effect of w_i on the network output, and therefore the loss, goes entirely through the summed input $z^{(t)}$ to the output unit. We can therefore evaluate the derivative of the loss with respect to w_i by appeal to chain rule

$$\frac{\partial}{\partial w_i} \text{Loss}\big(y^{(t)} z^{(t)}\big) \quad \overset{\text{chain rule}}{=} \quad \left[\frac{\partial z^{(t)}}{\partial w_i}\right] \left[\frac{\partial}{\partial z^{(t)}} \text{Loss}\big(y^{(t)} z^{(t)}\big)\right] \tag{22}$$

$$= \quad \left[\frac{\partial (\sum_{j=1}^d x_j^{(t)} w_j + w_0)}{\partial w_i}\right] \left[\frac{\partial}{\partial z^{(t)}} \text{Loss}\big(y^{(t)} z^{(t)}\big)\right] \tag{23}$$

$$= \quad \left[x_i^{(t)}\right] \left[\begin{array}{l} -y^{(t)} \text{ if } \text{Loss}\big(y^{(t)} z^{(t)}\big) > 0 \\ \text{and zero otherwise} \end{array}\right] \tag{24}$$

You may notice that $\text{Loss}(yz)$ is not actually differentiable at a single point where $yz = 1$. For our purposes here it suffices to use a *sub-gradient*[1] rather than a derivative at that point.

[1] Think of the knot at $yz = 1$ of $\max\{0, 1 - yz\}$ as a very sharp but smooth turn. Little changes in yz would change the derivative from zero (when $yz > 1$) to $-y$ (when $yz < 1$). All the possible derivatives around the point constitute a *sub-differential*. A *sub-gradient* is any one of them.

Put another way, there are many possible derivatives at $yz = 1$ and we opted for one of them (zero). SGD will work fine with sub-gradients.

We can now explicate SGD for the simple network. As promised, the updates look very much like perceptron or Pegasos. Indeed, we will get a non-zero update when $\text{Loss}\big(y^{(t)}z^{(t)}\big) > 0$ and in those cases

$$w_i \leftarrow w_i + \eta_k y^{(t)} x_i^{(t)}, \quad i = 1, \ldots, d \tag{25}$$

$$w_0 \leftarrow w_0 + \eta_k y^{(t)} \tag{26}$$

We can also initialize the parameters to all zero values as before. This won't be true for more complex models (as discussed later).

It remains to select the learning rate. We could just use a decreasing sequence of values such as $\eta_k = \eta_0/(k+1)$, as discussed above. This may not be optimal, however, as Figure 6 illustrates. In SGD, the magnitude of the update is directly proportional to the gradient (slope in the figure). When the gradient (slope) is small, so is the update. Conversely, if the objective varies sharply with θ, the gradient-based update would be large. But this is exactly the wrong way around. When the objective function varies little, we could/should make larger steps so as to get to the minimum faster. Moreover, if the objective varies sharply as a function of the parameter, the update should be smaller so as to avoid overshooting. A fixed and/or decreasing learning rate is oblivious to such concerns. We can instead adaptively set the step-size based on the gradients (AdaGrad):

$$g_i \leftarrow \frac{\partial}{\partial w_i}\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) \text{ (gradient or sub-gradient)} \tag{27}$$

$$G_i \leftarrow G_i + g_i^2 \text{ (cumulative squared gradient)} \tag{28}$$

$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{G_i}} g_i \text{ (adaptive gradient update)} \tag{29}$$

where the updates, as before, are performed for all the parameters, i.e., for all $i = 0, 1, \ldots, D$, in one go. Here η can be set to a fixed value since $\sqrt{G_i}$ reflects both the magnitude of the gradients as well as the number of updates performed. Note that we have some freedom here in terms of how to bundle the adaptive scaling of learning rates. In the example above, the learning rate is adjusted separately for each parameter. Alternatively, we could use a common scaling per node in the network such that all the incoming weights to a node are updated with the same learning rate, adjusted by the cumulative squared gradient that is now a sum of the individual squared derivatives.

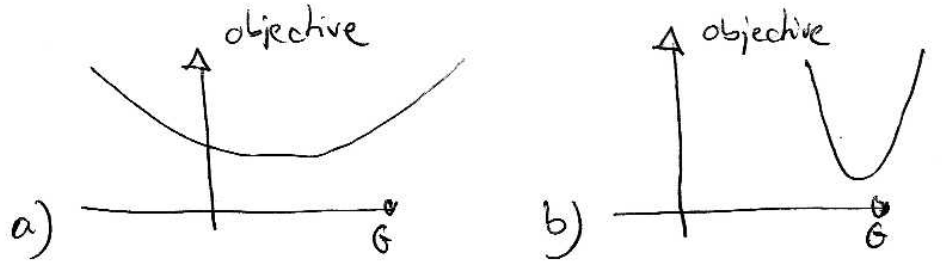

Figure 6: Objective functions that vary a) little b) a lot as a function of the parameter.

Two layer networks

Let us now consider the two layer network in Figure 3. Recall that the network output is now obtained in stages, activating the hidden units, before evaluating the output. In other words,

$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j} \quad \text{(input to the } j^{th} \text{ hidden unit)} \tag{30}$$

$$f(z_j) = \max\{0, z_j\} \quad \text{(output of the } j^{th} \text{ hidden unit)} \tag{31}$$

$$z = \sum_{j=1}^{m} f(z_j) w_j + w_0 \quad \text{(input to the last unit)} \tag{32}$$

$$F(x; \theta) = z \quad \text{(network output)} \tag{33}$$

The last layer (the output unit) is again simply a linear classifier but bases its decisions on the transformed input $[f(z_1), \ldots, f(z_m)]^T$ rather than the original $[x_1, \ldots, x_d]^T$. As a result, we can follow the SGD updates we have already derived for the single layer model. Specifically, when the loss is non-zero,

$$w_j \leftarrow w_j + \eta_k y^{(t)} f(z_j^{(t)}), \quad j = 1, \ldots, m \tag{34}$$

where $z_j^{(t)}$ is the input to the j^{th} hidden unit resulting from presenting $x^{(t)}$ to the network, i.e., $z_j^{(t)} = \sum_{i=1}^{d} x_i^{(t)} W_{ij} + W_{0j}$. Note that now $f(z_j^{(t)})$ serves the same role as the input coordinate $x_j^{(t)}$ did in the single layer network.

In order to update W_{ij}, we must consider how it impacts the network output. Changing W_{ij} will first change z_j, then $f(z_j)$, then finally z. In this case the path of influence of the parameter on the network output is unique. There are typically many such paths in multi-layer networks (and must all be considered). To calculate the derivative of the loss

with respect to W_{ij} in our case, we simply repeatedly apply the chain rule

$$\frac{\partial}{\partial W_{ij}}\text{Loss}\big(y^{(t)}z^{(t)}\big) = \left[\frac{\partial z_j^{(t)}}{\partial W_{ij}}\right]\left[\frac{\partial f(z_j^{(t)})}{\partial z_j^{(t)}}\right]\left[\frac{\partial z^{(t)}}{\partial f(z_j^{(t)})}\right]\left[\frac{\partial}{\partial z^{(t)}}\text{Loss}\big(y^{(t)}z^{(t)}\big)\right] \quad (35)$$

$$= [x_i^{(t)}]\,[\![\, z_j^{(t)} > 0 \,]\!]\,[w_j]\left[\begin{array}{l} -y^{(t)} \text{ if } \text{Loss}\big(y^{(t)}z^{(t)}\big) > 0 \\ \text{and zero otherwise} \end{array}\right] \quad (36)$$

Note that $f(z_j) = \max\{0, z_j\}$ is not differentiable at $z_j = 0$ but we will again just take a sub-gradient $[\![\, z_j > 0 \,]\!]$ which is one if $z_j > 0$ and zero otherwise. When there are multiple layers of units, the gradients can be evaluated efficiently by propagating them backwards from the output (where the signal lies) back towards the inputs (where the parameters are). This is because each previous layer must evaluate how the next layer affects the output as the influence of the associated parameters goes through the next layer. The process of propagating the gradients backwards towards the input layer is called *back-propagation*.

More concretely, let's see how we can cache some computations as we evaluate the derivatives. We proceed backwards, begin by evaluating and storing how the loss changes relative to just the input to the last output unit.

$$\delta^{(t)} = \frac{\partial}{\partial z^{(t)}}\text{Loss}\big(y^{(t)}z^{(t)}\big) = \left\{\begin{array}{l} -y^{(t)} \text{ if } \text{Loss}\big(y^{(t)}z^{(t)}\big) > 0 \\ \text{and zero otherwise} \end{array}\right. \quad (37)$$

Note that this value depends on the input $x^{(t)}$ as the derivative is evaluated at $z^{(t)}$ which is a function of $x^{(t)}$. Once we have $\delta^{(t)}$, we go back one layer and evaluate analogous derivatives for those units, i.e., how their inputs affect the overall network output. In other words,

$$\delta_j^{(t)} = \frac{\partial}{\partial z_j^{(t)}}\text{Loss}\big(y^{(t)}z^{(t)}\big) = \left[\frac{\partial f(z_j^{(t)})}{\partial z_j^{(t)}}\right]\left[\frac{\partial z^{(t)}}{\partial f(z_j^{(t)})}\right]\left[\frac{\partial}{\partial z^{(t)}}\text{Loss}\big(y^{(t)}z^{(t)}\big)\right] \quad (38)$$

$$= [\![\, z_j^{(t)} > 0 \,]\!]\,[w_j]\,\delta^{(t)} \quad (39)$$

for $j = 1, \ldots, m$. Note that we are using $\delta^{(t)}$ that was already calculated. Finally, the derivative with respect to the parameter W_{ij} that we were originally after can be expressed easily as a function of $\delta_j^{(t)}$

$$\frac{\partial}{\partial W_{ij}}\text{Loss}\big(y^{(t)}z^{(t)}\big) = \left[\frac{\partial z_j^{(t)}}{\partial W_{ij}}\right]\frac{\partial}{\partial z_j^{(t)}}\text{Loss}\big(y^{(t)}z^{(t)}\big) \quad (40)$$

$$= x_i^{(t)}\,\delta_j^{(t)} \quad (41)$$

So, put another way, by calculating δ's backwards, layer by layer, we can then easily evaluate the derivatives with respect to specific parameters on the basis of these cached values.

We are now ready to write down the simple SGD rule for W_{ij} parameters in the two layer network. Whenever $x^{(t)}$ isn't classified sufficiently well, i.e., $\text{Loss}\big(y^{(t)}z^{(t)}\big) > 0$,

$$W_{ij} \leftarrow W_{ij} - \eta_k\, x_i^{(t)}\delta_j^{(t)} i = 1, \ldots, d,\ j = 1, \ldots, m \quad (42)$$

The further back the parameters are, the more multiplications we have gone through in evaluating $\delta_j^{(t)}$. This has the effect that the δ's may easily either explode or vanish, precluding effective learning (known as exploding or vanishing gradient problem). The issue of learning rate is therefore quite important for these parameters but can be mitigated as discussed before (AdaGrad). Other "tricks" include just clipping the gradients as they propagate backwards so that, numerically, they don't explode.

Properly initializing the parameters is much more important in networks with two or more layers. For example, if we set all the parameters (W_{ij}'s and w_j's) to zero, then also $z_j^{(t)} = 0$ and $f(z_j^{(t)}) = 0$. Thus the output unit only sees an all-zero "input vector" $[f(z_1^{(t)}), \ldots, f(z_m^{(t)})]^T$ resulting in no updates. Similarly, the gradient for W_{ij} includes both $[\![z_j^{(t)} > 0]\!]$ and w_j which are both zero. In other words, SGD would forever remain at the all-zero parameter setting. Can you see why it would work to initialize W_{ij}'s to non-zero values while w_j's are set initially to zero?

Since hidden units (in our case) all have the same functional form, we must use the initialization process to break symmetries, i.e., help the units find different roles. This is typically achieved by initializing the parameters randomly, sampling each parameter value from a Gaussian distribution with zero mean and variance σ^2 where the variance depends on the layer (the number of units feeding to each hidden unit). For example, unit z_j receives d inputs in our two-layer model. We would like to set the variance of the parameters such that the overall input to the unit (after randomization) does not strongly depend on d (the number of input units feeding to the hidden unit). In this sense, the unit would be initialized in a manner that does not depend on the network architecture it is part of. To this end, we could sample each associated W_{ij} from a zero-mean Gaussian distribution with variance $1/d$. As a result, the input $z_j = \sum_{i=1}^d x_i W_{ij} + 0$ to each hidden unit (with zero offset) corresponds to a different random realization of W_{ij}'s. We can ask how $z_1, \ldots, z_m$ vary from one unit to another. This variance is exactly $(1/d)\sum_{i=1}^d x_i^2$ which does not scale with d.

Regularization, dropout training

Our network models can be quite complex (have a lot of power to overfit to the training data) as we increase the number of hidden units or add more layers (deeper architecture). There are many ways to regularize the models. The simplest way would be to add a squared penalty on the parameter values to the training objective, i.e., use SGD to minimize

$$\frac{1}{n}\sum_{t=1}^{n} \text{Loss}\big(y^{(t)} F(x^{(t)};\theta)\big) + \frac{\lambda}{2}\|\theta\|^2 \quad = \quad \frac{1}{n}\sum_{t=1}^{n} \overbrace{\left[\text{Loss}\big(y^{(t)} F(x^{(t)};\theta)\big) + \frac{\lambda}{2}\|\theta\|^2\right]}^{\text{modified loss}} \tag{43}$$

where λ controls the strength of regularization. We have rewritten the objective so that the regularization term appears together with each loss term. This way we can derive the SGD algorithm as before but with a modified loss function that now includes a regularization term. For example, w_j's in the two layer model would be now updated as

$$w_j \leftarrow w_j + \eta_k \bigg(-\lambda w_j + y^{(t)} f(z_j^{(t)}) \bigg), \quad j = 1, \ldots, m \tag{44}$$

when $\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) > 0$ and

$$w_j \leftarrow w_j + \eta_k\Big(-\lambda w_j + 0\Big), \;\; j = 1, \ldots, m \tag{45}$$

when $\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) = 0$. The additional term $-\lambda w_j$ pushes the parameters towards zero and this term remains even when the loss is zero since it comes about as the negative gradient of $\lambda\|\theta\|^2/2 = \lambda(\sum_{j=1}^m w_j^2 + \ldots)/2$.

Let's find a better way to regularize large network models. In the two layer model, we could imagine increasing m, the size of the hidden layer, to create an arbitrarily powerful model. How could we regularize this model such that it wouldn't (easily) overfit to noise in the training data? In order to extract complex patterns from the input example, each hidden unit must be able to rely on the behavior of its neighbors so to complement each other. We can make this co-adaption harder by randomly turning off hidden units. In other words, with probability 1/2, we set each hidden unit to have output zero, i.e., the unit is simply *dropped out*. This randomization is done anew for each training example. When a unit is turned off, the input/output weights coming in/going out will not be updated either. In a sense, we have dropped those units from the model in the context of the particular training example. This process makes is much harder for units to co-adapt. Instead, units can rely on the signal from their neighbors only if a larger numbers of neighbors support it (as about half of them would be present).

What shall we do at test time? Would we drop units as well? No, we can include all of them as we would without randomization. But we do need a slight modification. Since during training each hidden unit was present at about half the time, the signal to the units ahead is also about half of what it would be if all the hidden units were present. As a result, we can simply multiply the *outgoing* weights from each hidden unit by 1/2 to compensate. This works well in practice. Theoretical justification comes from thinking about the randomization during training as a way of creating large ensembles of networks (different configurations due to dropouts). This halving of outgoing weights is a fast approximation to the ensemble output (which would be expensive to compute).