Solutions for Test 1 for the Deep Learning Demystified class Fall 2026

1 Why do we need theoretical foundations?

Why should anyone care about theoretical explanations if we already have empirical successes? Theory is good when it leads to new results – but if results are already there, why care about explaining this success? If aspirin successfully cures a headache, why would anyone be interested in how exactly it works – which was, by the way, a mystery for many decades. There are two answers to this question.

First, when empirically, e.g., some activation function works better than others, this does not necessarily mean that this is indeed the best possible function: empirically, we can test only finitely many functions, and there are infinitely many of them. So, without a theoretical analysis, we cannot be sure that one of the functions that we did not try is actually better than the one we are using now. And maybe a theoretical analysis will find out that the empirically best function is not actually optimal – and this analysis will help us find a better function.

Second, theoretical results often apply not only to a specific application, but also to many other applications – including those that we have not yet empirically tried. This can provide us a guidance to solving other problems.

2 Why do we need machine learning?

One of the main objectives of science and engineering is to find out what is happening in the world, to predict – as accurately as possible – what will happen in the future, and to come up with the best way to influence this future. To solve all these problems, we need to process information.

To find out what is happening in the words, we need to use the observations and measurement results to estimate the values of the quantities which are not easy to observe or measure. For example, we would like to estimate the distances to a faraway star by measuring the angles describing its visual position at different seasons; we would like to estimate the amount of oil in an oil field by measuring local seismic signals and small local changes in gravity. We would

like to use the test results to diagnose potentially dangerous diseases as early as possible.

To predict the future state of the world, we need to estimate the future values of different quantities based on their current values. For example, we want to predict tomorrow's weather based on the meteorological data measured today and in the previous days.

To come up with the best way to influencing the future, we need to find the optimal values of the corresponding control parameters – i.e., the values that would lead to the most favorable future changes. For example, we may want to compute the trajectory of a ship from port A to port B that would lead to the smallest possible energy consumption (and thus, the smallest possible pollution). We may want to find the best medicine (and the best dosage) for curing a patient.

In all these cases, we want to estimate the value of some quantity y based on the known values of some other quantities $x_1, \ldots, x_n$.

In some cases, we know the algorithms needed to solve the corresponding data processing task – or at least we know the equations that describe the desired dependence. For example, to find the distance to a star based on the angle measurements, we can simply use the known algorithms for solving the corresponding system of trigonometric equations.

However, in many other cases, we do not know the exact form of the corresponding dependence. For example, while we may have a general idea of the medical diagnostics, for many diseases, we do not have a good understanding of how to make the corresponding diagnosis. We may have a general idea of how different doses of medicine affect a patient, but we usually do not have formulas or algorithms predicting the patient's reaction to different dosages.

In such situations, we need to extract the dependence between y and x_i from the observations. To be more precise, we have several cases when we observed both the values x_i and the desired value y. Based on these observations $\left(x_1^{(p)}, \ldots, x_n^{(p)}, y^{(p)}\right)$, $p = 1, 2, \ldots$, we need to find a dependence $y = f(x_1, \ldots, x_k)$ that fits all these observations, i.e., for which $y^{(p)} \approx f\left(x_1^{(p)}, \ldots, x_n^{(p)}\right)$ for all p. In the past, such dependence was found by people – this is how, e.g., Newton or Ohm came up with their laws. This is how we human learn many things: a kid observes several cats, several dogs, and learns to distinguish between cats and dogs. We see letters written in different fonts, in different handwriting – and we learn how to recognize these letters.

Nowadays, we also have computer programs that find such a dependence. Algorithms for finding the dependence based on observations are known as algorithms of *machine learning*.

3 Why neural networks?

In the old days, computers were much slower, this was a big limitation that prevented us from solving many important practical problems. As a result,

researchers started looking for ways to speed up computations.

If a person has a task which takes too long for one person, a natural idea is to ask for help, so that several people can work on this task in parallel – and thus, get the result faster. Similarly, if a computation task takes too long, a natural idea is to have several processing units working in parallel. In this case, the overall computation time is just to time that is needed for each of the processing unit to finish its sub-task.

To minimize the overall time, it is therefore necessary to make these sub-tasks as simple (and thus, as easy to compute) as possible. In data processing, the simplest possible functions to compute are linear functions. However, if we only have processing units that compute linear functions, we will only compute linear functions – since a composition of linear functions is always linear. Thus, we need to supplement these units with some nonlinear units. In general, the more inputs, the more complex (and thus longer) the resulting computations. So, the fastest possible nonlinear units are the ones that compute functions of one variable.

So, our ideal computational device should consist of linear (L) units and nonlinear units (NL) that compute functions of one variable. These units should work in parallel: first, all the units from one layer will work, then all units from another layer, etc. The fewer layers, the faster the resulting computations. One can prove that 1-layer schemes and even 2-layer schemes (both L followed by NL and NL followed by L) do not provide a universal approximation property, i.e., do not compute any given continuous function. One can also prove that 3-layer neurons already have this property. Among two possible 3-layer schemes: L-NL-L and NL-L-NL, the first one is faster, since it uses slower nonlinear units only once.

In this scheme, first, each unit from the first layer applies a linear transformation to the inputs $x_1, \ldots, x_n$, resulting in the values $z_k = \sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}$, for some values w_{ki} (known as *weights*). In the next NL layer, these values are transformed into $y_k = s_k(y_k)$, for some nonlinear functions $s_k(z)$. Finally, in the last L layers, the values y_k are linearly combined into the final result

$$y = \sum_{k=1}^{K} W_k \cdot y_k - W_0 = \sum_{k=1}^{K} W_k \cdot s_k\left(\sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}\right) - W_0.$$

This is exactly the formula that describes the traditional neural networks.

4 Why deep neural networks?

Traditional neural networks were invented when computers were reasonably slow, which prevented them from solving important practical problems. For these computers, computation speed was the main objective. As we have just shown, this need led to what we know as traditional neural networks.

Nowadays, computers are much faster. In most practical applications, speed is no longer the main problem. But now a new problem emerged: that the traditional neural networks, while fast, have limited accuracy of their predictions. It is therefore desirable to come up with devices that would achieve better prediction *accuracy* – even if to achieve this accuracy, they will be slower than the traditional neural networks. This is, in a nutshell, the main motivation behind what is now known as deep learning.

As a result of training a neural network – or any other machine learning tool – we get the values of some parameters for which the corresponding models provides the best approximation to the actual data. There are usually finitely many parameters, each of which – when represented in a computer – consists of finitely many bits. We can write down all these bits into a single binary sequence.

Let us denote the length of this sequence by N. If we have a parameters each of which consists of b bits, then $N = a \cdot b$.

In these terms, different models that can be obtained from training can be described by different N-bit sequences.

There are two possible values of each bit. For each of these values, we can have two different 2-but extensions depending on whether we extend the original bit with 0 or 1, so we have $2 \cdot 2 = 2^2$ possible sequences of length 2. For each of these 2-bit sequences, we have two possible 3-bits extensions, so the overall number of 3-bit sequences is $2 \cdot 2^2 = 2^3$. In general, for b bits, we have 2^b possible b-bit sequences.

In particular, if we use N bits to store all the information about all the tuned parameters, then we can have 2^N possible binary sequences and thus, 2^N possible models. In these terms, training simply means selecting one of these 2^N possible models.

If we have only one model to represent the actual dependence, this model will be a very lousy description. If we can have two models, we can have more accurate approximations. If we can have 100 different models, we can have an even more accurate one. In general, the more models we have, the more accurate representation we can have.

We can illustrate this idea on the example of approximating real numbers from the interval $[0, 1]$. If we have only one model – e.g., the value $x = 0.5$, then we approximate every other number with accuracy 0.5. If we can have 10 models, then we can take 10 values 0.05, 0.15, ..., 0.95. The first value approximates all the numbers from the interval $[0, 0.1]$ with accuracy 0.05. The second value approximates all the numbers from the interval $[0, 1, 0.2]$ with the same accuracy, etc. As a result, by selecting one these values, we can approximate any number from the interval $[0, 1]$ with accuracy 0.05.

If we can have 100 models, then can similarly take the values 0.005, 0.015, ..., all the way to 0.995 and this approximate any number from the interval $[0, 1]$ with accuracy 0.005.

How many models can we represent with a traditional neural network? Let us consider a traditional neural network with K neurons. Each neuron k is characterized by several weights W_k and w_{ki}. Let b denote the number of bits

needed to describe all the weights corresponding to a single neuron. Then, overall, to describe all possible bit sequences resulted from training, we need $N = K \cdot b$ bits.

As we mentioned, we can have 2^N different binary sequences of length N. So, at first glance, one may think that, in line with what we have discussed earlier, we can thus represent 2^N different models. However, the actual number of models is much smaller. The reason for this is that if we swap two neurons – i.e., swap all the bits corresponding to these two neurons – the resulting functions

$$f(x_1, \ldots, x_n) = \sum_{k=1}^{K} W_k \cdot s\left(\sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}\right) - W_0$$

will not change – since the sum does not change if we swap two of added numbers.

For example, if we have two neurons, the first one is described by a binary sequence 010 and the second one by 110, then this whole configuration is described by the sequence 010110. If we swap the neurons, we get a different binary sequence 110010 which, however, represents the same dependence – i.e., the same model.

Similarly, if instead of swapping two neurons, we apply any permutation, we get the exact same model.

For K neurons, there are $K!$ possible permutations.

Thus, $K!$ different binary sequences represent the same model. So, by using N neurons, instead of 2^N possible models, we can only have $\dfrac{2^N}{K!}$ possible models.

As we have mentioned, the more models we can represent, the more accurate will be the resulting approximation. When the overall number of bits is fixed – e.g., by the ability of our computers – the only way to increase the number of models is to decrease $K!$, i.e., to decrease K.

In the traditional neural networks, all the neurons are, in effect, in one layer – known as the hidden layer. The only way to decrease K is to make the number of neurons in each layer much smaller. This means that instead of placing the neurons into a single layer, we place then in many layers. Each of these layers is *narrow* – in the sense that the number of neurons in each later is small. We now have several layers – the construction is *deep* – so that the inputs go to the first layer, then the outputs of the first layer serve as inputs of the second layer, etc. – until we reach the final layer, which produces the resulting value y.

5 Why scale-invariance and shift-invariance?

Usually, we process the values of physical quantities, but the numerical values of physical quantities depend on what measuring unit we use and – for some quantities like temperature or time – what starting point we select for the measurement.

If we change a measuring unit to a one which is λ times smaller – e.g., replace feet with inches which are 12 times smaller – then all numerical values

get multiplied by λ – e.g., 2.5 feet becomes 30 inches. So, instead of the original numerical value x, we get a new numerical value $x' = \lambda \cdot x$.

Similarly, if we replace the original starting point with the new point which is x_0 units before, then each numerical value x is replaced by a new numerical value $x' = x + x_0$. For example, if instead of measuring years from our current Year 0 (supposedly the year when Jesus was born, although many historians disagree), we use Year -3761 (supposedly the year when the world was created, although here no historians agree), then current year 2019 becomes $2019 + 3761 = 5780$, which is our year in the traditional Jewish calendar.

We want to make sure that our data processing – in particular, our activation function – would not depend on the choice of a measuring unit.

6 Why ReLU: an explanation based on invariance

We want to select an activation function $s(x)$ that would not depend on the choice of a measuring unit. In other words, we want to make sure that if $y = s(x)$ and we select a new measuring unit, i.e., switch to new numerical values $x' = \lambda \cdot x$ and $y' = \lambda \cdot y$, then for these new values x' and y', we will have the exact same dependence:

$$y' = s(x').$$

Substituting the expressions $x' = \lambda \cdot x$ and $y' = \lambda \cdot y$ into this formula, we conclude that $\lambda \cdot y = s(\lambda \cdot x)$. Here, $y = s(x)$, so we conclude that $s(\lambda \cdot x) = \lambda \cdot s(x)$ for all possible x and $\lambda > 0$.

For $x = 1$, we conclude that $s(\lambda) = \lambda \cdot s(1)$. If we denote $s(1)$ by c_+, and rename λ into z, we conclude that for all $z > 0$, we get

$$s(z) = c_+ \cdot z.$$

For $x = -1$, we conclude that $s(-\lambda) = \lambda \cdot s(-1)$. If we denote $-s(-1)$ by c_- (so that $s(-1) = -c_-$) and denote $-\lambda$ by z (so that $\lambda = -z$), we conclude that for all negative values z, we have

$$s(z) = (-c_-) \cdot (-z) = c_- \cdot z.$$

Thus, we conclude that the activation function $s(z)$ should have the following *piecewise linear* form (see, e.g., [**?**]):

- for $z > 0$, we have $s(z) = c_+ \cdot z$;
- for $z < 0$, we have $s(z) = c_- \cdot z$.

Comment. We must have $c_+ \neq c_-$, since otherwise, the function $s(z)$ would be linear, and we know that with linear functions, we can only describe linear dependencies.

To uniquely determine a piecewise linear function, we need to select two real numbers: c_+ and c_-. The simplest possible real numbers is 0 and 1. Thus, the simplest possible piecewise linear function has the form:

- for $z > 0$, we have $s(z) = z$;
- for $z < 0$, we have $s(z) = 0$.

In other words, $s(z) = \max(z, 0)$. This function is known as *rectified linear function*, it is actually used in deep learning.

Why not use some other piecewise linear function? Because it does not matter much. Indeed, for all the layers – expect for the last one – the output of a neuron is linearly combined with other signals anyway. And any piecewise linear function can be represented as a linear combination of the rectified linear function $\max(z, 0)$ and the identity function z:

$$s(z) = c_- \cdot z + (c_+ - c_-) \cdot \max(z, 0).$$

Indeed:

- for $z > 0$, the right-hand side is equal to $c_- \cdot z + c_+ \cdot 0 = c_- \cdot z$, while
- for $z < 0$, the right-hand side is equal to

$$c_- \cdot z + (c_+ - c_-) \cdot z = (c_- + (c_+ - c_-)) \cdot z = c_+ \cdot z.$$

Why cannot we require shift-invariance instead of scale-invariance? We mentioned that the numerical value of a physical quantity changes when we change the measuring unit *and* when we change the starting point. However, when we looked for the activation function, we only considered invariance with respect to changing the unit (it is known as *scale-invariance*). Why not consider invariance with respect to changing the starting point (which is called *shift-invariance*)?

Let us give it a try. Similarly to the case of scale-invariance, we want to make sure that when $y = s(x)$ then for $x' = x + x_0$ and $y' = y + x_0$, we will have $y' = s(x')$. Substituting the expressions for x' and y' into the formula $y' = s(x')$, we get $y + x_0 = s(x + x_0)$. Here, $s(x) = y$, so we have $s(x + x_0) = s(x) + x_0$ for all possible values x and x_0.

In particular, for $x = 0$, we get $s(x_0) = s(0) + x_0$. Renaming $s(0)$ as a and x_0 as z, we conclude that $s(z) = z + a$. This is a linear function – thus, such neurons cannot describe any non-linear process.

7 Why ReLU: because it is the fastest to compute

In a computer, the fastest operations are the ones that are directly hardware supported. In most computers, the only directly hardware supported operations are arithmetic operations: min, max, addition/subtraction, multiplication, and

division. Also, shifts – which are equivalent to multiplying by a power of 2 – are easy to compute.

Whatever we want the computer to compute is actually implemented as a sequence of such elementary operations. For example, when we ask the computer to compute $\exp(x)$, it actually computes the polynomial approximation to $\exp(x)$ – that is equal to the sum of the first few terms in this function's Taylor expansion.

Out of these operations, the fastest are min and max, since these operations do not involve much data processing: they simply compare the two numbers and select one of them. Next fastest are addition/subtraction. Multiplication includes several additions – just like when we do it by hand – so it is somewhat slower. Similarly, division contains several multiplications – also similarly to how we do it by hand – so it is even slower.

From this viewpoint, the fastest-to-compute activation function $s(z)$ is the one that uses only min or max – of the input z and possible some constant c. The easiest-to-implement constant is usually 0, so we end up with $\max(0, z)$ and $\min(0, z)$. The first of these expressions is exactly ReLU, the second is equivalent to ReLU – if we take into account that between several application of the activation function, there are linear transformations. In our case, $\max(0, z) = -\min(0, -z)$. So, if we first apply a linear operation $z \mapsto -z$, then apply the $\min(0, z)$ activation function, and then again change the sign, we get exactly ReLU. Thus, whatever we can compute with ReLU neurons can be computed with the exact same number of neurons with activation function $\min(0, z)$. In other words, ReLU (and operations equivalent to ReLU) is optimal with respect to this optimality criterion – i.e., in this case, it is the fastest to compute.

8 Why invariance are important?

One of the main objectives of science and engineering is to predict future events – and to make sure that these future events are as beneficial to the humankind as possible. In general, what is the reason why we can make predictions in the first place?

For example, why are we sure that if we drop a pen (or any other object), it will start falling down with the acceleration of 9.81 m/sec^2? Because the same experiment was performed at different locations on the Earth, with the experimenters facing different directions, and in all these cases, we observed the same phenomenon. In other words, we can move to a different location, we can rotate – and this phenomenon will not change. In geometric terms, this phenomenon is symmetric with respect to shifts and rotations.

Such situations are ubiquitous. In all these situations, we have some transformations that do not change the corresponding physical process. In physics, such transformations are known as *symmetries.* In ordinary speech, we most apply this word to invariance under geometric transformations such as rotations or reflections in a plane, but in physics, this term is applied to any transforma-

tions, not necessarily geometric. For example, electromagnetic interactions will not change if we replace all positive charges with negative ones and vice versa – in physics terms, this is symmetry.

The idea of symmetry can be (and is) applied to even more complex situations than these. For example, how do we know that Ohm's law will hold when students do experiments in our lab? Because this law was followed on many different locations on Earth – and even on many locations outside the Earth surface.

Symmetries are one of the main tools in modern physics. Physicists fully understood the importance of symmetries around the 1960s, and this understanding revolutionized physics. Even the way physical theories are proposed has changed. In the past, starting from Newton, new physical theories were proposed in terms of differential equations. Nowadays, starting with quarks, new theories are described by listing the corresponding symmetries – and differential equations can be then derived from these symmetries. This is not only about new theories: it turned out that most fundamental physical theories that were proposed before this symmetries revolution – such as Maxwell's electrodynamics, quantum physics, special relativity, general relativity, etc. – can be reformulated in terms of the corresponding symmetries, symmetries that uniquely determine the corresponding differential equations.

Symmetries have been useful in computing as well. The same symmetries ideas have been applied to many application areas beyond physics. For example, it turned out that many empirically successful selection of intelligent techniques – be it selection of membership functions and "and"- and "or"-operations in fuzzy logic, selecting activation functions in neural networks, and appropriate functions in genetic algorithms and simulated annealing – all this can be naturally explained by the corresponding symmetries. As for modern deep neural networks with their spectacular successes – most of their empirically successful features can also be explained by symmetries.

9 Why optimal alternatives are invariant?

In general, what do we mean by an optimality criterion, i.e., by a criterion that allows us to select one of many possible alternatives? In many cases, we have a well-defined objective function $F(a)$ – i.e., we have a numerical value $F(a)$ attached to each alternative a. We then select the alternative a for which this value is – depending on what we want – either the largest or the smallest.

For example, when we look for the shortest path:

- we assign, to each path a, its length $F(a)$, and
- we select the path for which this length is the smallest possible.

When we look for an algorithm for solving problems of given size, often:

- we assign, to each algorithm a, the worst-case computation time $F(a)$ on all inputs of this size, and

- we select the algorithm a for which this worst-case time $F(a)$ is the smallest possible.

However, an optimality criterion can be more complicated. For example, we may have several different shortest paths a for a car to go from one city location to another. In this case, it may be reasonable to select, among these shortest paths, a path a along which the overall exposure to pollution $G(a)$ is the smallest. The resulting optimality criterion can no longer be described by a single objective function, it is more complicated: we prefer a to a' if:

- either $F(a) < F(a')$
- or $F(a) = F(a')$ and $G(a) < G(a')$.

Similarly, if we have two different algorithms a with the same worst-case computation time $F(a)$, we may want to select, among them, the one for which the average computation time $G(a)$ is the smallest possible. In this case too, we prefer a to a' if:

- either $F(a) < F(a')$,
- or $F(a) = F(a')$ and $G(a) < G(a')$.

The optimality criterion can be even more complicated. However, no matter how many different objective functions we use, we do need to have a way to compare different alternatives. Thus, we can define a general optimality criterion as a relation $\preceq$ on the set of all possible alternatives, so that $a \preceq a'$ means that the alternative a' is better (or of the same quality) than the alternative a.

Naturally, each alternative has the same quality as itself, so we should have $a \preceq a$. Such relations are called *reflexive*. Also, if a' is better than or of the same quality as a and a'' is better than or of the same quality as a', then clearly a'' should be better than or of the same quality as a. relations with this property are known as *transitive*. So, we arrive at the following definition.

Definition 1. *Let a set A be given; its elements will be called alternatives.*

- *We say that a relation $\preceq$ on the set A is* reflexive *if $a \preceq a$ for all a.*
- *We say that a relation $\preceq$ is* transitive *if $a \preceq a'$ and $a' \preceq a''$ imply that $a \preceq a''$.*
- *By an* optimality criterion *in the set A, we mean a reflexive transitive relation $\preceq$ on this set.*

What is an optimal alternative? In these terms, an alternative a_{opt} is *optimal* if is better (or of the same quality) than all other alternatives.

Definition 2. *We say that an alternative a_{opt} is* optimal *with respect to the optimality criterion $\preceq$ on the set A if $a \preceq a_{\text{opt}}$ for all $a \in A$.*

The optimality criterion must be useful. We want an optimality criterion to be useful, i.e., we want to use it to select an alternative. Thus, there should be at least one alternative which is optimal according to this criterion.

Definition 3. *We say that an optimality criterion $\preceq$ is* useful *if there exists at least one optimal alternative.*

When is the optimality criterion final? What if several different alternatives are optimal according to the given criterion? In this case, we can use this non-uniqueness to optimize something else.

For example, if on a given class of benchmarks, neurons that use several different activation functions have the same average approximation error, we can select, among the, the function with the smallest computational complexity.

This way, instead of the original optimality criterion, we, in effect, use a new criterion according to which s_0 is better than s_0 if:

- either it has the smaller average approximation error
- or it has the same average approximation error and smaller computational complexity.

If, based on this modified criterion, we still have several different activation functions which are equally good, we can use this non-uniqueness to optimize something else: e.g., worse-case approximation accuracy, etc.

Thus, every time the optimality criterion selects several equally good alternatives, we, in effect, replace it with a modified criterion, and keep modifying it until *finally* we get a criterion for which only one alternative is optimal. So, we arrive at the following definition.

Definition 4. *We say that a useful optimality criterion is* final *if there exists exactly one alternative which is optimal with respect to this optimality criterion.*

Symmetries: general description. In many practical situations, we have transformations that do not affect the physics of the situation.

Definition 5. *Let a set A be a given.*

- *A* transformation *is a reversible function from A to A.*
- *A set G of transformations is called a* transformation group *if it satisfies the following two properties:*
  - *if $g \in G$, then the inverse transformation g^{-1} should also belong to G, and*
  - *if f and g belong to G, then their composition $f \circ g$ also belongs to G.*
- *Transformations from the group G are called* symmetries.

For natural transformations, the relation $a \preceq a'$ should not change if we apply the same symmetry to both a and a'. This leads to the following definition.

Definition 6. *We say that the optimality criterion $\preceq$ is* invariant *with respect to the transformation group G (or simply G-*invariant*, for short) if for all $a, a' \in A$ and for all $g \in G$, we have*

$$a \preceq a' \Leftrightarrow g(a) \preceq g(a').$$

Main lemma. It turns out that for invariant final optimality criteria, the optimal alternative is also invariant, in some reasonable sense.

Definition 7. *We say that an alternative $a_0 \in A$ is* invariant *with respect to a transformation group G (or, for short, G-*invariant*) if $g(a_0) = a_0$ for all $g \in G$.*

Proposition 1. *For each G-invariant final optimality criterion, the optimal alternative $a_{\rm opt}$ is also G-invariant.*

Proof of Proposition 1. Let $a_{\rm opt}$ be the optimal alternative, and let g be any transformation from the group G. We want to prove that $g(a_{\rm opt}) = a_{\rm opt}$. To prove this equality, let us prove that the alternative $g(a_{\rm opt})$ is optimal. Then, the desired equality will follow from the fact that the optimality criterion is final – and thus, there is only one optimal alternative.

To prove that the alternative $g(a_{\rm opt})$ is optimal, we need to prove that $a \preceq g(a_{\rm opt})$ for all a. Due to G-invariance of the optimality criterion, this condition is equivalent to $g^{-1}(a) \preceq a_{\rm opt}$ – which is, of course, always true, since $a_{\rm opt}$ is optimal. Thus, $g(a_{\rm opt})$ is also optimal, hence $g(a_{\rm opt}) = a_{\rm opt}$ for all $g \in G$. The proposition is proven.

10 Which dependencies are scale-invariant?

Scale-invariance means for each $\lambda > 0$, there should exist a value $\mu > 0$ such that:

- if $y = f(x)$,
- then $y' = f(x')$, where $x' \stackrel{\rm def}{=} \lambda \cdot x$ and $y' \stackrel{\rm def}{=} \mu \cdot y$.

Which dependencies are scale-invariant? Substituting $y' = \mu \cdot y$ and $x' = \lambda \cdot x$ into the formula $y' = f(x')$, we get $\mu \cdot y = f(\lambda \cdot x)$. Here, we have $y = f(x)$, so $f(\lambda \cdot x) = \mu \cdot f(x)$. Taking into account that μ depends on λ, we get the following expression:

$$f(\lambda \cdot x) = \mu(\lambda) \cdot f(x). \qquad (*)$$

We can conclude that the function $\mu(\lambda)$ is equal to the ratio of two differentiable functions $\mu(\lambda) = \dfrac{f(\lambda \cdot x)}{f(x)}$ and is, thus, differentiable too.

Since both functions $f(x)$ and $\mu(\lambda)$ are differentiable, we can differentiate both sides of the formula (*) with respect to λ. After plugging in $\lambda = 1$, we get

$$x \cdot \frac{df}{dx} = a \cdot f,$$

where we denoted

$$a \stackrel{\text{def}}{=} \frac{d\mu}{d\lambda}_{|\lambda=1}.$$

We can separate the variables in this formula if we divide both sides by f and by x, then we get:

$$\frac{df}{f} = a \cdot \frac{dx}{x}.$$

Integrating both sides, we get $\ln(f) = a \cdot \ln(x) + C$, where C is the integration constant. Applying the exponential function to both sides of this formula, we get $f = c \cdot x^a$, where we denoted $c \stackrel{\text{def}}{=} \exp(C)$. So, every scale-invariant dependence is a power law.

11 Which dependencies are shift-invariant?

Shift-invariance means for each x_0, there should exist a value $\mu > 0$ such that:

- if $y = f(x)$,
- then $y' = f(x')$, where $x' \stackrel{\text{def}}{=} x + x_0$ and $y' \stackrel{\text{def}}{=} \mu \cdot y$.

Which dependencies are shift-invariant? Substituting $y' = \mu \cdot y$ and $x' = x + x_0$ into the formula $y' = f(x')$, we get $\mu \cdot y = f(x + x_0)$. Here, we have $y = f(x)$, so $f(x) = \mu \cdot f(x)$. Taking into account that μ depends on x_0, we get the following expression:

$$f(\lambda \cdot x) = \mu(x_0) \cdot f(x). \qquad (**)$$

since the function $f(x)$ is differentiable, then the function $\mu(x_0) = \dfrac{f(x + x_0)}{f(x)}$ is differentiable too. Thus, we can differentiate both sides of the equation (**) with respect to x_0. As a result, we get

$$f'(x + x_0) = \mu'(x_0) \cdot f(x). \qquad (A1)$$

In particular, for $x_0 = 0$, we get

$$\frac{df}{dx} = a \cdot f,$$

where $a \stackrel{\text{def}}{=} \mu'(0)$. We can separate the variables x and f if we multiply both sides of this equality by $\dfrac{dx}{f}$, then we get

$$\frac{df}{f} = a \cdot dx.$$

Integrating both sides, we get

$$\ln(f) = a \cdot x + C,$$

where C is the integration constant. Applying the function $\exp(z)$ of both sides of the equality (A4), we get the desired expression $f(x) = A \cdot \exp(a \cdot x)$, with $A = \exp(C)$.

12 Why softmax?

Formulation of the problem. We want to describe how the probability $p(a)$ of selecting an alternative a depends on the value $J(a)$) of the corresponding objective function. In other words, we want to find a non-decreasing function $f(x)$ for which the probability $p(a)$ is proportional to $f(J(a))$, i.e., for which $p(a) = c \cdot f(J(a))$. Since the probabilities add up to 1, we have $\sum\limits_{a \in A} p(a) = c \cdot \sum\limits_{a \in A} f(J(a)) = 1$, hence $c = \dfrac{1}{\sum\limits_{a \in A} f(J(a))}$ and

$$p(a) = \frac{f(J(a))}{\sum\limits_{a' \in A} f(J(a'))}.$$

One can easily see that if we multiply all the values of the function $f(x)$ by the same constant, the probabilities remain the same. Since we are interested only in the probabilities, this means that instead of selecting a single function $f(x)$, we should select a *family* of functions $\{c \cdot f(x)\}_c$, where a function $f(x)$ is given, and c takes all possible positive values.

Definition. *By a* family*, we mean a family of functions $\{c \cdot f(x)\}_c$, where $f(x)$ is a given non-decreasing function and c takes all possible positive values.*

What are natural symmetries here? The main purpose of selecting an objective function is to decide which alternative is better and which is worse. If we add a constant to all the values of the objective function, this does not change which ones are better and which ones are worse. Thus, it is reasonable to require that the optimality criterion on the class of all the families should not change if we simply add a constant x_0 to all the values x, i.e., replace each function $f(x)$ with $f(x + x_0)$.

Definition. *For each x_0, by a* x_0-shift *of a family $\{c \cdot f(x)\}_c$, we mean the family $\{c \cdot f(x + x_0)\}_c$.*

Proposition 2. *For each final shift-invariant optimality criterion on the set of all families, the optimal family corresponds to $f(x) = \exp(\beta \cdot x)$ for some $\beta \geq 0$.*

Discussion. This result explains why this particular version of softmax has been most empirically successful.

Proof. Due to Proposition 1, the optimal family should be shift-invariant. This means, in particular, that if we take the function $f(x)$ from the original optimal family and shift it, then the resulting function $f(x + x_0)$ should also belong to the same optimal family, i.e., we should have $f(x + x_0) = c(x_0) \cdot f(x)$ for some c depending on x_0.

We have shown that the only non-decreasing solutions to this functional equation are functions $f(x) = \text{const} \cdot \exp(\beta \cdot x)$ for some $\beta > 0$. The proposition is proven.

13 Why pooling?

Often, we have a lot of data points to process. For example, an image is usually represented in a computer by the intensities at each pixel. Even for a not very good 1000 by 1000 picture, we have 1,000,000 pixels – so to process such an image, we need to process 1,000,000 numbers.

In a traditional neural network, we could use as many neurons as needed, but in a deep neural network, there are only a few neurons in the first layer. Thus, before we start processing, we need to combined several input values into one (a similar procedure can also be applied at a later stage). This operation of combining several values into one is known as *pooling*.

14 Which poolings should we use?

Let us consider the case when we pool two values a and b into a single value c. Let us denote the resulting value c by $p(a, b)$. Of course, the pooling should not depend on the order, i.e., we should have $p(a, b) = p(b, a)$. In other words, the pooling operation should be *commutative*.

Similarly to the above discussions, it is reasonable to require that the result of pooling will not change if we simply change the measuring unit or change the starting point for measurement. So, we make two requirements:

- if $c = p(a, b)$, then $c' = p(a', b')$, where $a' = \lambda \cdot a$, $b' = \lambda \cdot b$, and $c' = \lambda \cdot c$;
- if $c = p(a, b)$, then $c' = p(a', b')$, where $a' = a + a_0$, $b' = b + a_0$, and $c' = c + a_0$.

From the first requirement, substituting the expressions $a' = \lambda \cdot a$, $b' = \lambda \cdot b$, and $c' = \lambda \cdot c$ into the formula $c' = p(a', b')$, we conclude that $\lambda \cdot c = p(\lambda \cdot a, \lambda \cdot b)$. Here, $c = p(a, b)$, so we conclude that

$$p(\lambda \cdot a, \lambda \cdot b) = \lambda \cdot p(a, b).$$

From the second requirement, substituting the expressions $a' = a + a_0$, $b' = b + a_0$, and $c' = c + a_0$ into the formula $c' = p(a', b')$, we conclude that $c + a_0 = p(a + a_0, b + a_0)$. Here, $c = p(a, b)$, so we conclude that $p(a + a_0, b + a_0) = p(a, b) + a_0$.

Let us use the resulting formulas to find the value $p(x, y)$ for all possible pairs (x, y). Without losing generality, we can assume that $x < y$. Then, substituting $a = 0$, $a_0 = x$, and $b = y - x$ into the formula $p(a + a_0, b + a_0) = p(a, b) + a_0$, we conclude that

$$p(x, y) = p(0, y - x) + x.$$

Substituting $\lambda = y - x$, $a = 0$, and $b = 1$ into the formula $p(\lambda \cdot a, \lambda \cdot b) = \lambda \cdot p(a, b)$, we conclude that $p(0, y - x) = (y - x) \cdot p(0, 1)$. Substituting this expression into the formula $p(x, y) = p(0, y - x) + x$ and denoting $p(0, 1)$ by α, we conclude that

$$p(x, y) = x + \alpha \cdot (y - x) = \alpha \cdot y + (1 - \alpha) \cdot x.$$

Once we learn how to pool two values, we can pool four values easily:

- divide the four values into two pairs,
- pool results within each pair, and then
- pool the two pooling results into a single value.

It is reasonable to require that the result should not depend on how exactly we divide the four original values into pairs. Let us consider the values 0, 1, 1, and 2.

First, we combine 0 with 1 and 1 with 2. Pooling 0 and 1 results in $\alpha \cdot 1 + (1 - \alpha) \cdot 0 = \alpha$. Pooling 1 and 2 results in

$$\alpha \cdot 2 + (1 - \alpha) \cdot 1 = 2\alpha + 1 - \alpha = 1 + \alpha.$$

Here always $1 + \alpha$ is larger than α, so combining the results α and $1 + \alpha$ leads to

$$\alpha \cdot (1 + \alpha) + (1 - \alpha) \cdot \alpha = \alpha + \alpha^2 + \alpha - \alpha^2 = 2\alpha.$$

What if we instead combine 1 with 1 and 0 with 2? Combining 1 with 1 results in $\alpha \cdot 1 + (1 - \alpha) \cdot 1 = 1$. Pooling 0 with 2 results in

$$\alpha \cdot 2 + (1 - \alpha) \cdot 0 = 2\alpha.$$

The resulting of pooling the resulting too values 1 and 2α depends on which of the two values is larger.

- If $2\alpha \geq 1$, i.e., if $\alpha \geq 0.5$, then we get

  $$\alpha \cdot (2\alpha) + (1 - \alpha) \cdot 1 = 2\alpha^2 + 1 - \alpha.$$

  In this case, the desired equality is $2\alpha^2 + 1 - \alpha = 2\alpha$, i.e.,

  $$2\alpha^2 - 3\alpha + 1 = 0.$$

  One can easily check that this quadratic equation has two solutions: $\alpha = 0.5$ and $\alpha = 1$.

- If $2\alpha \le 1$, i.e., if $\alpha \le 0.5$, then we get

$$\alpha \cdot 1 + (1-\alpha) \cdot 2\alpha = \alpha + 2\alpha - 2\alpha^2 = 3\alpha - 2\alpha^2.$$

  In this case, the desired equality is $3\alpha - 2\alpha^2 = 2\alpha$, i.e.,

$$2\alpha^2 - \alpha = 0.$$

  One can easily check that this quadratic equation has two solutions: $\alpha = 0$ and $\alpha = 0.5$.

So, we have three options: $\alpha = 0$, $\alpha = 0.5$, and $\alpha = 1$.

- If $\alpha = 0$, then the pooling formula takes the form $p(x, y) = 0 \cdot y + 1 \cdot x = x$, i.e., since $x \le y$, the form

$$p(x, y) = \min(x, y).$$

- If $\alpha = 0.5$, then the pooling formula takes the form $p(x, y) = 0.5 \cdot y + 0.5 \cdot x$, i.e., to the arithmetic average

$$p(x, y) = \frac{x + y}{2}.$$

- If $\alpha = 1$, then the pooling formula takes the form $p(x, y) = 1 \cdot y + 0 \cdot x = y$, i.e., since $x \le y$, the form

$$p(x, y) = \max(x, y).$$

These three operations – minimum, maximum, and arithmetic averge – are indeed the ones which work most successfully in deep learning.