C. bend those lines to capture rich patterns (units 2,3)

shallow learning

By the end of this section, you'll be able to

- train (and make predictions using) a shallow binary classifier
- visualize learned features and decision boundaries for shallow nets
- derive and intuitively interpret shallow nets' learning gradients

Thanks to lpamarescot for help!

So many learners volunteered to help write notes. Alas, it turns out there are weird legal barriers to sharing our LaTeX source code, so my original idea of sharing latex source that y'all could improve will not happen. Another challenge has been more on me: I've been running behind on drafting notes to begin with!

A pale but still-meaningul substitute: *if you have any suggestions for changes to the notes, please tell me!*

A MENU OF FEATURIZATION FUNCTIONS — We've discussed linear models in depth. We've seen how important it is to prepare the data for linear models by choosing appropriate featurizations — for example, applying the $(x \mapsto (1, x))$ bias trick can drastically improve testing accuracy! So we've *hand-coded* non-linearities to extract usable features from raw features. This makes our models more expressive.

Now we'll discuss how to *learn* features from data.° This idea is called 'deep learning'. The word 'deep' references that soon we will layer feature-learners on top of each other. But we'll start simple, with just one feature-learning 'layer'.

← We came close to this when talking about kernel methods. Kernel methods use a featurization that depends on the training inputs. But, intuitively, that featurization isn't particularly 'fitted' (e.g. we haven't chosen our kernels by gradient descent on some interesting loss). Less importantly, the featurizations we used when using kernels don't depend on the training outputs.

Let's build a logistic classifier that *learns* the features it ultimately separates-via-hyperplane. In brief, the classifier will be described by a weight matrix A that combines those features just as in Unit 1, together with a weight matrix B that *defines* those features in terms of the raw input. The numeric entries of A, B change during training; as B changes, the features it defines change to become more useful, and this is what we mean when we say that we "learn features". We'll call this classifier a **shallow neural network**. Note, however, that those two matrices' numbers of columns and of rows stay the same during training; we specify those shapes as part of our design process. So, while we no longer handcraft features (i.e., do manual feature selection), we still choose how many features to use and what the "allowed shapes" for each feature are. Those choices are part of **architecture**. We can tune architectural choices as we do other hyperparameters, for instance by cross-validation.

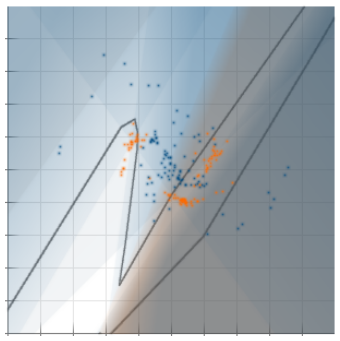


Figure 23: A toy example of the decision boundary (**black**) of a shallow neural network on 2D inputs (preprocessed with the bias trick). This neural network has 8 features (shown as subtle discontinuities in shading, with less shading when that feature is negative and more shading when that feature is positive), and we depict the weight on each feature by the shading's saturation. The next couple pages explain how we build a model that learns such features and can have such decision boundaries.

The above is our roadmap. Let's see how B actually appears in our math. We want the classifier to learn a featurization function that maps each an input x, represented via raw (or 'rawer') features, to some representation $\tilde{x}$ more useful to the task. That is, we will present to the machine a menu $\{\cdots, \varphi, \cdots\}$ of possible featurization functions and we want the machine to select a particular function φ to use as a way to translate raw features x to features $\tilde{x}$.

What menu should we use? Well, we've already seen (hardware and theory) advantages in defining a menu by giving a function φ_B for each matrix B, where $\varphi_B(x)$ somehow relates to the product Bx. But φs must be non-linear in order to increase expressivity. So let's process Bx through a nonlinear function f:

$$\tilde{x} = f(Bx)$$

We define our menu of possible featurizing functions as the set of functions of the above form. The whole menu shares the same f; menu items differ in their Bs.

What f shall we use? Commonly used fs include the ReLU function $\mathrm{relu}(z) = \max(0, z)$ and variants.° But keep in mind that we often encounter situations where domain-specific knowledge suggests special fs other than ReLU variants. Anyway, let's use the "leaky" ReLU variant $\mathrm{lrelu}(z) = \max(z/10, z)$. Actually, for z an array we will do that operation on each component separately, and we'll throw in the bias trick:

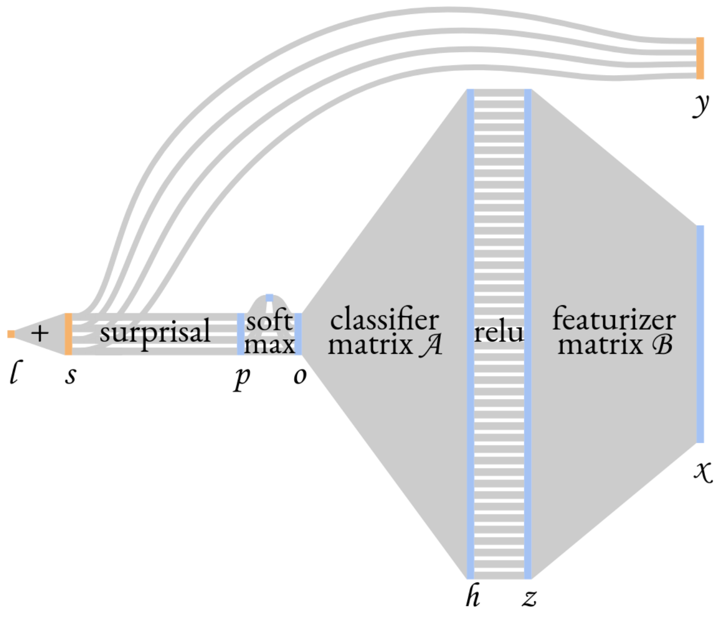
←Older projects often use functions like $f(x) = \tanh(x)$. We'll see such functions show up playing specialized roles (e.g. in LSTMs, which themselves are now a bit outdated for their main use case in language). But experience has now shown that these functions aren't a good default choice.

$$f(z[0], z[1], \cdots) = (1, \max(z[0]/10, z[0]), \max(z[1]/10, z[1]), \cdots)$$

So overall our logistic classifier represents the probability model:

$$\hat{p}(y{=}{+}1\,|\,x) \;=\; \sigma(A(f(Bx)))$$

We have a hypothesis for each (A, B) pair. Here A is our familiar weight vector that linearly separates features $\tilde{x}$; what's new is that a B influences how $\tilde{x}$ depends on x. We want to learn both A and B from data.



+
surprisal
soft max
classifier matrix $\mathcal{A}$
relu
featurizer matrix $\mathcal{B}$
ℓ
s
p
o
y
x
h
z

Figure 24: **Architecture of shallow neural nets.** Data flows right to left via gray transforms. We use the blue quantities to predict; the orange, to train. Thin vertical strips depict vectors; small squares, scalars. We train the net to maximize data likelihood per its softmax predictions:

$$\ell = \textstyle\sum_k s_k \quad s_k = y_k \log(1/p_k)$$
$$p_k = \exp(o_k)/\textstyle\sum_{\tilde{k}} \exp(o_{\tilde{k}})$$

The decision function o is a linear combination of features h nonlinearly transformed from x:

$$o_k = \textstyle\sum_j A_{kj} h_j$$

Each "**hidden activation**" or "**learned feature**" h_j measures tresspass past a linear boundary determined by a vector B_j:

$$h_j = \mathrm{lrelu}(z_j) = \max(z_j/10, z_j) \quad z_j = \textstyle\sum_i B_{ji} x_i$$

We've not depicted a bias term but you should imagine it present.

Let's recap while paying attention to array sizes. Our logistic classifier is:

$$\hat{p}(y{=}{+}1\,|\,x) \;=\; (\sigma_{1\times 1} \circ A_{1\times(h+1)} \circ f_{(h+1)\times h} \circ B_{h\times d})(x)$$

where A, B are linear maps with the specified $(\text{input} \times \text{output})$ dimensions, where σ is the familiar sigmoid operation, and where f applies the leaky relu function elementwise and concatenates a 1:

$$f((v_i : 0 \le i < h)) = (1,) +\!\!+ (\mathrm{lrelu}(v_i) : 0 \le i < h) \qquad \mathrm{lrelu}(z) = \max(z/10, z)$$

We call h the **hidden dimension** of the model. Intuitively, $f \circ B$ re-featurizes the input to a form more linearly separable (by weight vector A).

ACTIVATION FUNCTIONS — Here's a brief aside on why we use activation functions such as lrelu. *I don't want to overemphasize this topic*, even though it is important, since it's best appreciated once you've gotten your hands dirty with code, and we don't want to miss the forest for the trees on our journey to that code-writing stage.

Food For Thought: Sketch $\text{lrelu}(z), \text{relu}(z), \tanh(z)$ against z. See $\text{lrelu}(z)$'s two linear pieces with different slopes; its derivative is never very close to 0 and this eliminates one of the major ways that gradient descent can get "stuck", namely, (to use a physics analogy) the way a ball can get stuck atop a mesa's flat plateau instead of falling down the cliff. This is the so-called **vanishing gradient** problem. *This problem does not affect our Unit 1 linear models. Why is this?*

Food For Thought: Here's a toy example of vanishing gradients. Let's use a model $f_{a,b,c}(x) = a + b\tanh(c + x)$, with a, b, c, x all numbers.° We initialize $(a, b, c) = (0, 0)$ and run gradient descent (GD) with least-squares loss on three datapoints $(x, y) = (-60, -1), (-40, -1), (-20, +1)$. This data is well-explained by $\theta_\star = (a, b, c) = (0, 1, 30)$. But the weights take a very long time to get near $\theta_\star$. Do you see why? What if we use relu instead of tanh? *What if we use lrelu?*

← You'll recognize a, b as analogous to the matrix A in our above architecture and c as analogous to the matrix B.

Especially before ~2018, and especially in deep, "dynamic" models such as RNNs and GANs, the vanishing gradient problem was severe. Nowadays we use techniques such as "batch normalization", "adaptive gradients", and lrelu to cure the vanishing gradients problem.

TRAINING BY GRADIENT DESCENT — Gradient descent works the same:

$$w \leftarrow w - \eta \nabla \ell(w)$$

where $w = (A, B)$ consists of all learned parameters (here, the coeffiecients of both A and B), ℓ is the loss on a training batch, and η is the learning rate.

We've already learned how to compute $d\ell/do$ and $d\ell/dA = (h)(d\ell/do)^T$ (to use the notation of the architecture figure). Likewise we may compute $d\ell/dh = A^T(d\ell/do)$. To address the nonlinearity we use the chain rule:

$$d\ell/dz_k = (d\ell/dh_k) \cdot \text{lrelu}'(z_k) \qquad \text{lrelu}'(z_k) = (1/10) \text{ if } z_k < 0 \text{ else } 1$$

and finally, in strict analogy to $d\ell/dA = (h)(d\ell/do)^T$, our B-gradient $d\ell/dB = (x)(d\ell/dz)$. This process of working backward using the product rule and chain rule is called **backpropagation** — it's an organized system for computing derivatives efficiently.

Intuitively, $d\ell/dA$ tells us how to re-weigh the features we have while $d\ell/dB$ tells us how to change our features. The image I have in mind is of shifting pressure between one's legs vs sliding one's feet across the floor.

One more thing — **To break symmetry, we should initialize with (small) random weights rather than at zero. Do you see why?**

Let's see what these gradient dynamics look like. Our decision boundaries look more complicated, as expected. We also depict each learned feature.

Next 3 questions: we initialize $A = B = 0$ and work qualitatively/roughly.

Food For Thought: What is the training loss at initialization?

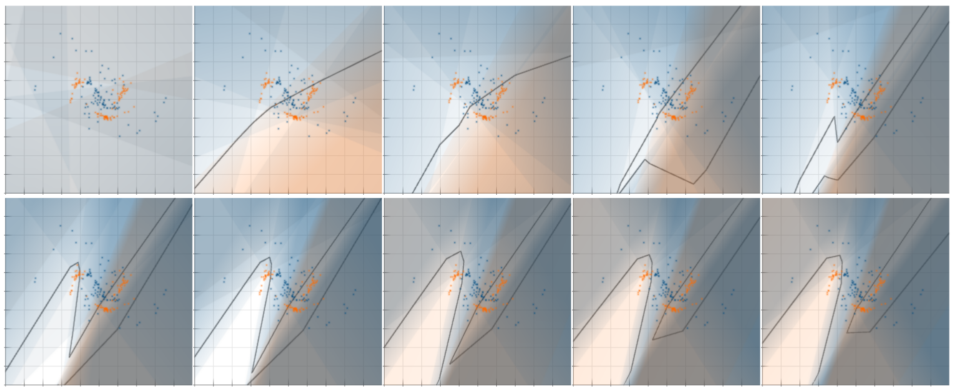


Figure 25: **Training dynamics of a shallow neural net**. We use artificial 2D data. The net has 8 hidden features, which we depict as subtle edges between colors. In bold are overall decision boundaries. In English reading order (left-to-right top row, then left-to-right bottom row), we show the network after 0, 500, 1000, etc many gradient steps. **Notice the features 'swinging around' to better capture the patterns in the data.**

Food For Thought: What is the loss gradient at initialization?

Food For Thought: What is the testing accuracy after a thousand SGD updates?

For the questions below, assume a fixed, smallish training set and a fixed, moderate number of gradient descent steps. Work qualitatively/roughly.

Food For Thought: what should the training and testing accuracies look like as a function of hidden dimension?

Food For Thought: what should the training and testing accuracies look like as a function of the learning rate?

REGULARIZATION — Here is a simple way to generalize L2 regularization for shallow neural networks:

$$\text{regularization penalty} = \lambda_A \|A\|^2 + \lambda_B \|B\|^2$$

Here, $\|A\|^2, \|B\|^2$ are the sums of the squares of the entries of those matrices. This leads to the gradient terms

$$A^{\text{new}} = A - \eta \cdot (\cdots + 2\lambda_A A) \qquad\qquad B^{\text{new}} = B - \eta \cdot (\cdots + 2\lambda_B B)$$

Here, the $\cdots$ represent the gradients from the non-regularization terms.

It is often a good idea to regularize bias coefficients by a different amount — potentially by 0 — than the λs used for the other weights.

Food For Thought: Observe that L2 regularization disambiguates the scaling redundancy coming from our choice to use lrelu activations. What I mean by "scaling redundancy" is that if we change (A, B) into $(A', B') = (A/68, 68B)$, then we get the same predictions: $A \cdot \text{lrelu}(B \cdot x) = A' \cdot \text{lrelu}(B' \cdot x)$.

The $\|A\|^2$ term favors *large margins with respect to the learned features*, just as we saw for linear models. When the $\|B\|^2$ term is small, large margins with respect to the learned features imply *large margins with respect to the raw inputs*. The two terms work together; if we just used one of them, the aforementioned scaling redundancy would allow the network to have small margins with respect to raw inputs.

As we move toward deep learning, we will start focusing on "implicit" or "architectural" methods of regularization as supplements to and even replacements for the explicit regularization as above. You can also google "batch normalization" or (for historical interest but a bit outdated) "dropout". Google stuff and then ask questions in the forum; I'll try to answer.

A TOY EXAMPLE — Here's a toy dataset that can help develop a mental model for shallow neural networks. It's binary classification of 3-D input vectors within the cube $[-1,+1] \times [-1,+1] \times [-1,+1]$. We'll write the components of each vector as (x_0, x_1, x_2). The points all obey $x_2 = \epsilon \cdot \text{sign}(\min(x_0, x_1))$ for $\epsilon = 1/10$. The points are classified as positive or negative according to whether their x_2 is positive or negative. The data is uniformly distributed across the described region.

So this dataset is linearly separable. Nevertheless, we want to classify it using a shallow neural network with 2 hidden features. For simplicity, we use hinge loss on the values $Af(Bx)$ instead of logistic loss. Also for simplicity, we do the bias trick neither on the raw inputs nor on the learned features. So A has shape 1×2 and B has shape 2×3. We constrain A and the two rows of B to be unit vectors; this models the effect of regularization by keeping all weights small-ish. Food For Thought: The shallow neural net can mimic a linear classifier by setting $A = [-1, 0]$ and $B = [[0, 0, -1], [1, 0, 0]]$. *In which direction will gradient descent tilt A and B?* And, after taking a few gradient steps, *how does the shallow neural network's decision boundary change from a linear hyperplane?* Think qualitatively, not quantitatively.

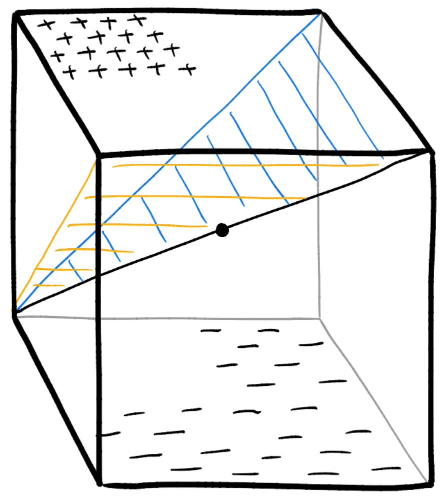


Figure 26: An illustration of the training data (black +s and −s) and of a "butterfly" shaped decision boundary that the shallow neural network can express. We defined the training data with $\epsilon = 1/10$ (+s and −s closer together vertically) but here we depict the training data with $\epsilon = 1$ to make the figure easier to understand.

TOWARD DEEP NEURAL NETWORKS — Inspired by the idea of turning raw inputs into more useful features, we can iterate, turning those learned features into even more useful learned features:

$$\hat{p}(y=+1\,|\,x) = (\sigma_{1\times 1} \circ A_{1\times(h+1)} \circ f_{(h+1)\times h} \circ B_{h\times(\tilde{h}+1)} \circ f_{(\tilde{h}+1)\times\tilde{h}} \circ C_{\tilde{h}\times d})(x)$$

Here, the matrix C turns x into a feature vector of dimension $\tilde{h}$, then the matrix B turns *that* into a feature vector of dimension h, then the matrix A classifies based on those super-duper learned features. In jargon, we say that this model has "2 hidden activation layers" (counting those two fs) or "3 weight layers" (couting to those 3 matrices). And we can keep going. Models with dozens of layers are now the norm; models with hundreds of layers have been successfully explored. More jargon: the number of layers is **depth**; the dimension of each layer is **width**.°

← Once you understand how to derive gradient descent for the shallow case, deriving gradient descent for deeper networks will feel easy.

Both depth and width increase our model's complexity. They do so in different ways. Insofar as specific values for weight matrices define a "program" that transforms inputs to outputs, large *depth* allows the definition of helper functions in terms of primitives while large *width* allows the direct combination of many primitives.

By analogy, we might say that a lecturer has "expressive grammar" if they combine words into phrases, phrases into clauses, clauses into sentences, sentences into paragraphs, in intricate combinatorial patterns full of nuance. We might say a lecturer has "expressive vocabulary" if they use just-right, not-so-common words to vividly capture their meaning. Depth allows complex grammar. Width allows complex vocabulary.

For example, imagine a NN for processing images of what we have in our fridge. Its first layer features can detecting basic color patterns (such as streaks of red, leafy edges, shadows of something bulky, etc) but not for detecting "higher" concepts such as *red apple* vs *red cherry*.

The second layer starts with these basic color patterns as inputs, so it can build in complexity. It can detect fruit parts by how those basic color patterns "hang together": if a small leafy edge appears directly above a small shadowy region, then it is more likely to be a true 3d leaf rather than a 2d sticker that happens to be green; if a small red streak appears next to a bright white segment, then it is more likely to be part of a beef-atop-styrofoam package than a patch of ripe apple.

The layer after that takes as input these fruit-part measurements — one of those inputs will be high when seems to be a leaf near the image's center; another of those will be high when there seems to be a patch of apple near the image's left. By combining these inputs, it can detect actual fruits and distinguish far-away red apples from nearby cherries, even though both look like red disks of the same sizes.◦

Neural nets are good at squeezing as much as they can out of correlations in the training data.◦ So, if we are training an apple-vs-cherry classifier, and if in our training examples apples tend slightly to co-occur with crammed fridges◦ and cherries tend slightly to co-occur with sparse fridges, then the neural net will pick up on this. It will learn not just what local color patterns look more like cherries or apples but also what global color patterns look more like crammed fridges vs sparse ones. Sometimes the network's learning of such a feature is desirable; other times, not.

But this packet of notes is supposed to just lay the groundwork for Unit 3. I'll try to discuss more about depth, CNNs, and RNNs in future packets of notes!

← By the way, the story in these paragraph is optimistic. It is true as long as we are okay with mediocre (but still much better than chance) accuracies. In practice, unless the image data is especially simple, one should try more layers for computer vision.

← In this whole section, I'm making general claims about usual architectures trained by usual methods. I am not claiming universal laws.

← Perhaps because the kind of fridge-user who refrigerates apples tends to refrigerate all of their food. In the cold climate of Michigan, where I'm from, we usually don't have to refrigerate apples.