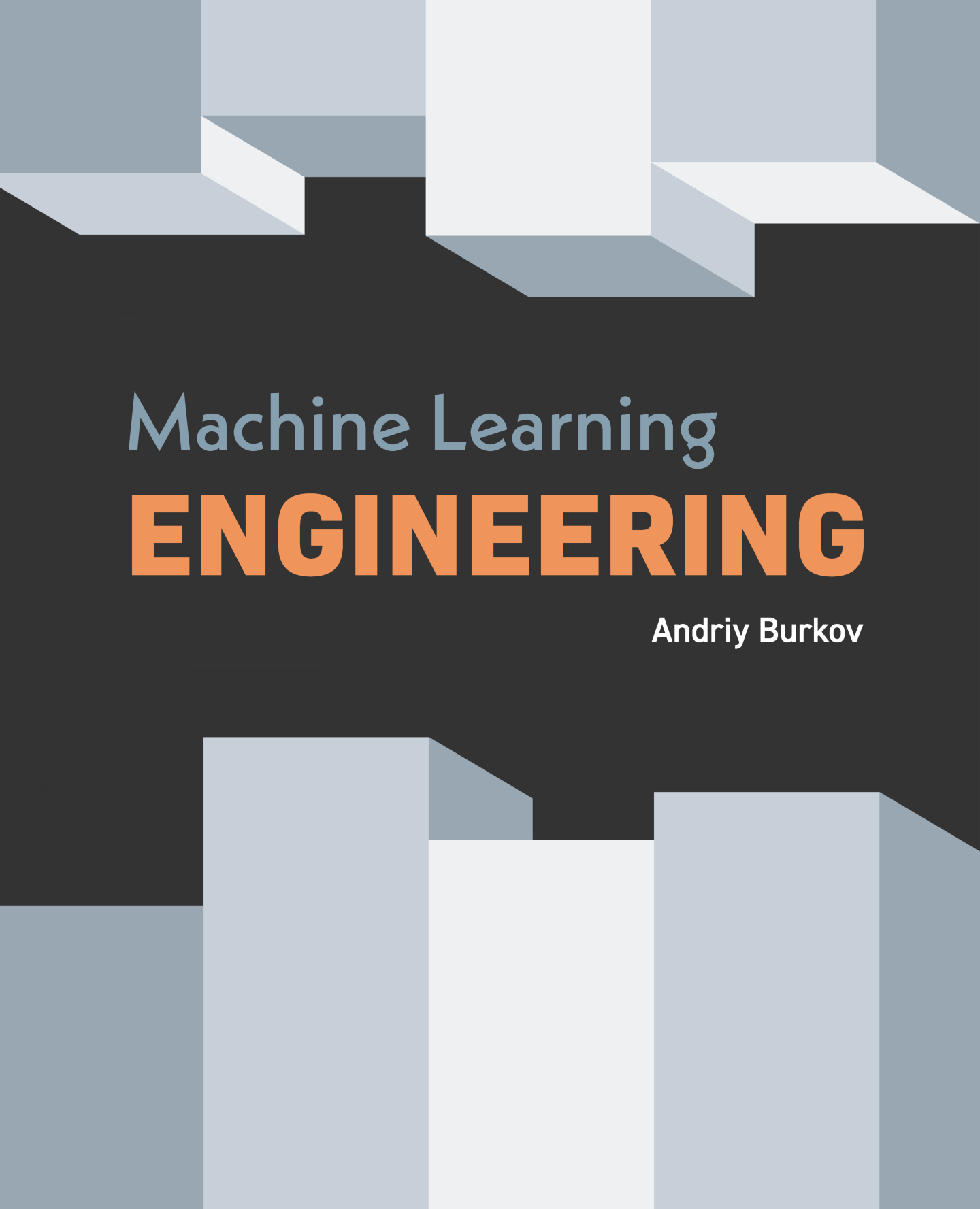

Machine Learning
ENGINEERING
Andriy Burkov

“In theory, there is no difference between theory and practice. But in practice, there is.”

— Benjamin Brewster

“The perfect project plan is possible if one first documents a list of all the unknowns.”

— Bill Langley

“When you’re fundraising, it’s AI. When you’re hiring, it’s ML. When you’re implementing, it’s linear regression. When you’re debugging, it’s printf().”

— Baron Schwartz

The book is distributed on the “read first, buy later” principle.

6 Supervised Model Training (Part 2)

In this second part of our conversation about supervised model training, we consider such topics as training deep models, stacking models, handling imbalanced datasets, distribution shift, model calibration, troubleshooting and error analysis, and other best practices.

Compared to shallow models, the model training strategy for deep neural networks has more moving parts. On the other hand, it's more principled and better amenable to automation.

6.1 Deep Model Training Strategy

Model training starts with shortlisting several network architectures, also known as **network topologies**. If you work with image data and you want to build your model from scratch, then a **convolutional neural network** (CNN) with at least one **convolutional layer**, followed by a **max-pooling layer**, and one **fully connected layer** may be your default topology choice.

If you work with text or other sequence data, such as time series, you have a choice between a CNN, a **gated recurrent neural network** (such as Long Short Term Memory, **LSTM**, or gated recurrent units, **GRU**), or a **Transformer**.

Instead of training your model from scratch, you can also start with a **pre-trained model**. Companies like Google and Microsoft have trained very deep neural networks with architectures optimized for image or natural language processing tasks.

Among the most used pre-trained models for image processing tasks are **VGG16** and **VGG19** (based on the Visual Geometry Group, **VGG**, architecture), **InceptionV3** (based on the **GoogLeNet** architecture), and **ResNet50** (based on the **residual network** architecture).

For natural language text processing, such pre-trained models as Bi-directional Encoder Representations from Transformer, **BERT**, (based on the Transformer architecture) and Embeddings from Language Models, **ELMo** (based on the **bi-directional LSTM** architecture) often improve the quality of the model, compared to training a model from scratch.

An advantage of using pre-trained models is that these were trained on huge quantities of data available to its creators, but likely unavailable to you. Even if your dataset is smaller and not exactly similar to the one used to pre-train the model, the parameters learned by the pre-trained models may still be useful.

You can use a pre-trained model in two ways:

1) use its learned parameters to initialize your own model, or
2) use the pre-trained model as a feature extractor for your model.

If you use the pre-trained model the former way, it gives you more flexibility. The downside is you end up training a very deep neural network. That requires significant computational

resources. In the latter case, you "freeze" the parameters of the pre-trained model and only train the parameters of added layers.

6.1.1 Neural Network Training Strategy

Using an existing model to create a new model is called **transfer learning**. We will talk more on this topic in Section 6.1.10. For the moment, assume you are building a model from scratch, based on the architecture of your choice. A common strategy to build a neural network looks as follows:

1. Define a performance metric P.
2. Define the cost function C.
3. Pick a parameter-initialization strategy W.
4. Pick a cost-function optimization algorithm A.
5. Choose a hyperparameter tuning strategy T.
6. Pick a combination H of hyperparameter values using the tuning strategy T.
7. Train model M, using algorithm A, parametrized with hyperparameters H, to optimize cost function C.
8. If there are still untested hyperparameter values, pick another combination H of hyperparameter values using strategy T, and repeat step 7.
9. Return the model for which the metric P was optimized.

Now let's discuss some of the steps of the above strategy in detail.

6.1.2 Performance Metric and Cost Function

Step 1 is similar to step 1 of the shallow model training strategy (Section **??**): we define a metric that would allow comparing the performance of two models on the holdout data, and select the better of the two. An example of a performance metric is **F-score** or **Cohen's kappa**.

In step 2, we define what our learning algorithm will optimize in order to train a model. If our neural network is a regression model, then, in most cases, the cost function is the **mean squared error** (MSE) defined in Equation **??** in the previous chapter. Let's repeat it here:

$$\text{MSE}(f) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{i=1\ldots N} (f(\mathbf{x}_i) - y_i)^2.$$

For classification, a typical choice for the cost function is either **categorical cross-entropy** (for multiclass classification) or **binary cross-entropy** (for binary and multi-label classification).

Recall that when we train a neural network for **multiclass classification**, we should represent labels using the **one-hot encoding**. Let C be the number of classes in our classification

problem. Let $\mathbf{y}_i$ be a one-hot encoded label of example i, where i spans from 1 to N. Let $y_{i,j}$ denote the value in position j (where j spans from 1 to C) in example i. The categorical cross-entropy loss for classification of example i is defined as,

$$\text{CCE}_i \stackrel{\text{def}}{=} -\sum_{j=1}^{C} \left[y_{i,j} \times \log_2(\hat{y}_{i,j}) \right],$$

where $\hat{\mathbf{y}}_i$ is the C-dimensional vector of prediction issued by the neural network for the input $\mathbf{x}_i$. The cost function is typically defined as the sum of losses of individual examples:

$$\text{CCE} \stackrel{\text{def}}{=} \sum_{i=1}^{N} \text{CCE}_i \,.$$

In **binary classification**, the output of the neural network for the input feature vector $\mathbf{x}_i$, is a single value $\hat{y}_i$, while the label of the example is a single value y_i, just like in logistic regression. The binary cross-entropy loss for classification of example i is defined as,

$$\text{BCE}_i \stackrel{\text{def}}{=} -y_i \times \log_2(\hat{y}_i) - (1 - y_i) \times \log_2(1 - \hat{y}_i).$$

Similarly, the cost function for classification of the training set is typically defined as the sum of losses of individual examples:

$$\text{BCE} \stackrel{\text{def}}{=} \sum_{i=1}^{N} \text{BCE}_i \,.$$

Binary cross-entropy is also used in **multi-label classification**. The labels are now C-dimensional **bag-of-words** vectors $\mathbf{y}_i$, while the predictions are C-dimensional vectors $\hat{\mathbf{y}}_i$, whose values $\hat{y}_{i,j}$ in each dimension j range between 0 and 1. The loss for the prediction of one label $\hat{\mathbf{y}}_i$ is defined as,

$$\text{BCEM}_i \stackrel{\text{def}}{=} \sum_{j=1}^{C} \left[-y_{i,j} \times \log_2(\hat{y}_{i,j}) - (1 - y_{i,j}) \times \log_2(1 - \hat{y}_{i,j}) \right].$$

The cost function for the classification of the entire training set is typically defined as the sum of losses of individual examples,

$$\text{BCEM} \stackrel{\text{def}}{=} \sum_{i=1}^{N} \text{BCEM}_i \,.$$

Note that the output layers in multiclass and multi-label classification are different. In multiclass classification, one **softmax** unit is used. It generates a C-dimensional vector whose values are bounded by the range $(0, 1)$, and whose sum equals 1. In multi-label classification, the output layer contains C logistic units whose values also lie in the range $(0, 1)$, but their sum lies in the range $(0, C)$.

Neural Network Output

The curious reader may wish to better understand the logic behind choosing a specific loss function. This block will mathematically describe the output of a neural network.

In regression, the output layer contains only one unit. If the output value can be any number, from minus infinity to infinity, then the output unit will not contain non-linearity. On the other hand, if the neural network must predict a positive number, then the **ReLU** (rectified linear unit) non-linearity can be used. Let the output value of the output unit before non-linearity for the input example i be denoted as z_i. Then the output after applying the ReLU non-linearity is given by $\max(0, z_i)$.

In a binary classification, the output layer contains only one logistic unit. Let the output value of the output unit before non-linearity for the input example i be denoted as z_i. The output $\hat{y}_i$ after applying the logistic nonlinearity is given by,

$$\hat{y}_i \stackrel{\text{def}}{=} \frac{1}{1 + e^{-z_i}},$$

where e is the base of the natural logarithm, also known as **Euler's number**.

Binary and multi-label classification models are defined in a similar way. The only difference is that in multi-label classification, the output layer contains C logistic units, one per class. If $\hat{y}_{i,j}$ denotes the output, after nonlinearity, of the logistic unit for class j, when input example is i, then the sum of $\hat{y}_{i,j}$, for all $j = 1, \ldots, C$, lies between 0 and C.

In the multiclass classification, the output layer also produces C outputs. However, in this case, the output of each unit of the output layer is controlled by the softmax function. Let the output of the output unit j, before nonlinearity, for the input example i, be $z_{i,j}$. Then the output $\hat{y}_{i,j}$ after nonlinearity is given by,

$$\hat{y}_{i,j} \stackrel{\text{def}}{=} \frac{e^{z_{i,j}}}{\sum_{k=1}^{C} e^{z_{i,k}}}.$$

The sum of $\hat{y}_{i,j}$, for all $j = 1, \ldots C$, equals 1.

6.1.3 Parameter-Initialization Strategies

In step 3, we select a **parameter-initialization strategy**. Before the training starts, the parameter values in all units are unknown. We must initialize them with some values. Training algorithms for neural networks, such as **gradient descent** and its stochastic variants that we consider in a few moments, are iterative in nature and require the analyst to specify some initial point from which to begin the iterations. This initialization might affect the properties of the training model. You will likely choose from one of these strategies:

- **ones** — all parameters are initialized to 1;
- **zeros** — all parameters are initialized to 0;
- **random normal** — parameters are initialized to values sampled from the **normal distribution**, typically with mean of 0 and standard deviation of 0.05;
- **random uniform** — parameters are initialized to values sampled from the **uniform distribution** with the range $[-0.05, 0.05]$;
- **Xavier normal** — parameters are initialized to values sampled from the truncated normal distribution, centered on 0, with standard deviation equal to $\sqrt{2/(\text{in} + \text{out})}$ where "in" is the number of units in the preceding layer to which the current unit is connected (the one whose parameters you initialize); and "out" is the number of units on the subsequent layer to which the current unit is connected; and,
- **Xavier uniform** — parameters are initialized to values sampled from a uniform distribution within $[-\text{limit}, \text{limit}]$, where "limit" is $\sqrt{6/(\text{in} + \text{out})}$, and "in" and "out" are defined as in Xavier normal, above.

There are other initialization strategies. If you work with a neural network training module such as TensorFlow, Keras, or PyTorch, they provide some parameter initializers, and also recommend default choices.

The bias term is usually initialized with a zero.

While we know the parameter initialization affects the model properties, we cannot predict which strategy will provide the best result for your problem. Random and Xavier initializers are the most common. It's recommended to start your experiments with one of those two.

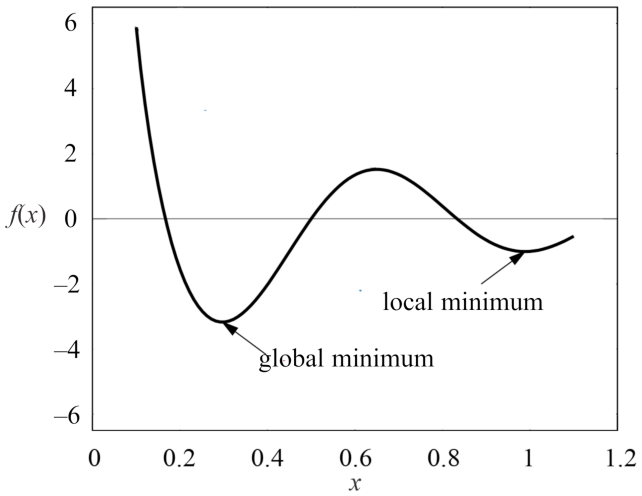


Figure 1: A local and a global minima of a function.

6.1.4 Optimization Algorithms

In step 4, we select a cost-function optimization algorithm. When the cost function is differentiable (and it's the case for all cost functions we considered above) **gradient descent** and **stochastic gradient descent** are two most frequently used optimization algorithms.

Gradient descent is an iterative optimization algorithm for finding a **local minimum** of any differentiable function. We say that $f(x)$ has a **local minimum** at $x = c$ if $f(x) \geq f(c)$ for every x in some **open interval** around $x = c$. An **interval** is a set of real numbers with the property that any number that lies between two numbers in the set is also included in the set. An open interval does not include its endpoints and is denoted using parentheses. For example, $(0, 1)$ means "all numbers greater than 0 and less than 1." The minimal value among all the local minima is called the **global minimum**. The difference between a local and a global minimum of a function is shown in Figure 1.

Functions and optimization

> In this block, for the curious reader, we explain the basics of mathematical function and function optimization. If you only want to know the mechanics of training neural networks, you can safely skip it.
>
> A **function** is a relation that associates each element x of a set $\mathcal{X}$, the **domain** of the function, to a single element y of another set $\mathcal{Y}$, the **codomain** of the function. A function usually has a name. If the function is called f, this relation is denoted $y = f(x)$, read "y equals f of x." The element x is the argument, or input of the

function, and y is the value of the function, or the output. The symbol that is used for representing the input is the variable of the function. We often say that f is a function of the variable x.

A **derivative** f' of a function f is a function or a value that describes how fast f increases or decreases. If the derivative is a constant value, like 5 or -3, then the function increases or decreases constantly, at any point x of its domain. If the derivative f' is itself a function, then the function f can grow at a different pace in different regions of its domain. If the derivative f' is positive at some point x, then the function f increases at this point. If the derivative of f is negative at some x, then the function decreases at this point. The derivative of zero at x means that the function neither decreases nor increases at x; the function's slope at x is horizontal.

The process of finding a derivative is called **differentiation**.

Derivatives for basic functions are known. For example if $f(x) = x^2$, then $f'(x) = 2x$; if $f(x) = 2x$ then $f'(x) = 2$; if $f(x) = 2$ then $f'(x) = 0$. The derivative of any function $f(x) = c$, where c is a constant value, is zero.

If the function we want to differentiate is not basic, we can find its derivative using the **chain rule**. For instance if $F(x) = f(g(x))$, where f and g are some functions, then $F'(x) = f'(g(x))g'(x)$. For example if $F(x) = (5x + 1)^2$ then $g(x) = 5x + 1$ and $f(g(x)) = (g(x))^2$. By applying the chain rule, we find $F'(x) = 2(5x + 1)g'(x) = 2(5x + 1)5 = 50x + 10$.

Gradient is the generalization of derivatives for functions that take several inputs, or one input in the form of a vector or some other complex structure. A gradient of a function is a vector of **partial derivatives**. Finding a partial derivative of a function is the process of finding the derivative by focusing on one of the function's inputs and considering all other inputs as constant values.

For example, if our function is defined as $f([x^{(1)}, x^{(2)}]) = ax^{(1)} + bx^{(2)} + c$, then the partial derivative of function f with respect to $x^{(1)}$, denoted as $\frac{\partial f}{\partial x^{(1)}}$, is given by,

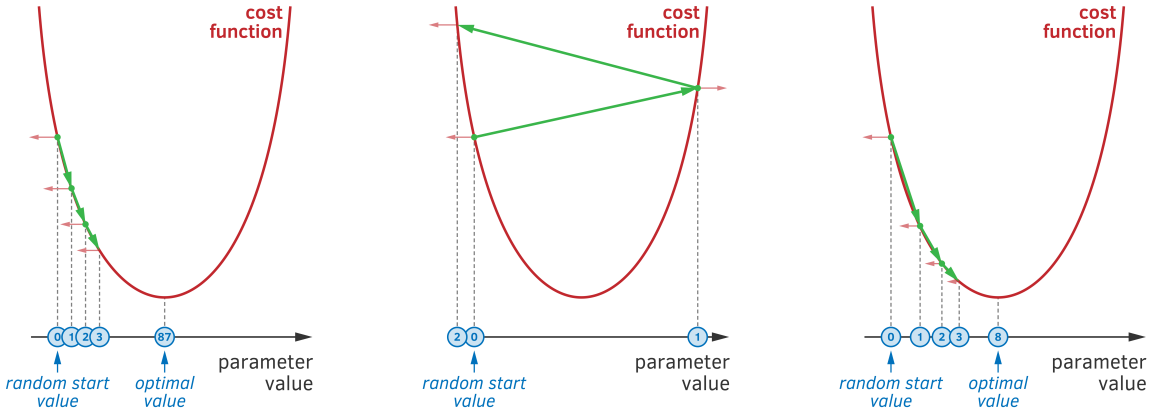
$$\frac{\partial f}{\partial x^{(1)}} = a + 0 + 0 = a,$$

where a is the derivative of the function $ax^{(1)}$. The two zeros are respectively derivatives of $bx^{(2)}$ and c, because $x^{(2)}$ is considered constant when we calculate the derivative with respect to $x^{(1)}$, and the derivative of any constant is zero.

Similarly, the partial derivative of function f with respect to $x^{(2)}$, $\frac{\partial f}{\partial x^{(2)}}$, is given by,

$$\frac{\partial f}{\partial x^{(2)}} = 0 + b + 0 = b.$$

The gradient of function f, denoted as ∇f is given by the vector $[\frac{\partial f}{\partial x^{(1)}}, \frac{\partial f}{\partial x^{(2)}}]$.



(a) too small learning rate (b) too large learning rate (c) good learning rate

Figure 2: The influence of the learning rate on the convergence: (a) too small, the convergence will be slow; (b) too large, no convergence; (c) the right value of learning rate.

The chain rule works with partial derivatives too.

To find a local minimum of a function using **gradient descent**, we start at some random point in the domain of the function. Then we move proportionally to the negative of the gradient (or approximate gradient) of the function at the current point.

Gradient descent in machine learning proceeds in **epochs**. An epoch consists of using the training set entirely to update each parameter. In the first epoch, we initialize the parameters of our neural network using one of the parameter-initialization strategies discussed above. The **backpropagation** algorithm computes the partial derivatives of each parameter using the chain rule for derivatives of complex functions.[1] At each epoch, gradient descent updates all parameters using partial derivatives. The **learning rate** controls the significance of an update. The process continues until **convergence**, the state when the values of parameters don't change much after each epoch. Then the algorithm stops.

Gradient descent is sensitive to the choice of the learning rate α. Picking the right learning rate for your problem is not easy. If you select a value that is too high, you might not reach convergence at all. On the other hand, too small values of α can slow down the learning to the point of no observable progress. In Figure 2, you can see an illustration of gradient descent for one parameter of a neural network and three values of the learning rate. The value of the parameter at each iteration is shown as a blue circle. The number inside the

[1]The explanation of backpropagation is beyond the scope of this book. You should only know that every modern software library for training neural networks contains an implementation of this algorithm. The curious reader can find the explanation of backpropagation in the extended version of The Hundred-Page Machine Learning Book on its companion wiki.

circle indicates the epoch. The red arrows indicate the direction of the gradient along the horizontal axis — the direction away from the minimum. The green arrows show the change in the value of the cost function after each epoch.

Therefore, at each epoch, gradient descent moves the parameter value towards the minimum. If the learning rate is too small, the movement towards the minimum will be very slow (Figure 2a). If the learning rate is too large, the value of the parameter will oscillate away from the minimum (Figure 2b).

Gradient descent is rather slow for large datasets because it uses the entire dataset to compute the gradient of each parameter at each epoch. Fortunately, several significant improvements to this algorithm have been proposed.

Minibatch stochastic gradient descent (minibatch SGD) is a variant of the gradient descent algorithm. It approximates the gradient using small subsets of the training data called **minibatches**. This effectively speeds up the computation. The size of the minibatch is a hyperparameter, and you can tune it. Powers of two, between 32 and a few hundred, are recommended: 32, 64, 128, 256, and so on.

The problem of choosing a value for the learning rate α is still present in the "vanilla" minibatch SGD. Learning can still stagnate at later epochs. Instead of reaching a local minimum, the gradient descent might keep oscillating around it due to too large updates. There are many **learning rate decay schedules** that allow updating the learning rate, as the learning progresses, by reducing it later in the epoch count. The benefits of using a learning rate decay schedule include faster gradient descent convergence (faster learning) and higher model quality. Below, we consider several popular learning rate decay schedules.

6.1.5 Learning Rate Decay Schedules

Learning rate decay consists of gradually reducing the value of the **learning rate** α as the epochs progress. Consequently, the parameter updates become finer. There are several techniques, known as schedules, to control α.

Time-based learning rate decay schedules alter the learning rate depending on the learning rate of the previous epoch. The mathematical formula for the learning rate update, according to a popular time-based learning rate decay schedule, is:

$$\alpha_n \leftarrow \frac{\alpha_{n-1}}{1 + d \times n},$$

where α_n is the new value of the learning rate, α_{n-1} is the value of the learning at the previous epoch $n-1$, and d is the **decay rate**, a hyperparameter. For example, if the initial value of the learning rate $\alpha_0 = 0.3$, then the values of the learning rate at the first five epochs are shown in the table below:

learning rate	epoch
0.15	1
0.10	2
0.08	3
0.06	4
0.05	5

Step-based learning rate decay schedules change the learning rate according to some pre-defined drop steps. The mathematical formula for the learning rate update, according to a popular step-based learning rate decay schedule, is:

$$\alpha_n \leftarrow \alpha_0 d^{\text{floor}\left(\frac{1+n}{r}\right)},$$

where α_n is the learning rate at epoch n, α_0 is the initial value of the learning rate, d is the decay rate that reflects how much the learning rate should change at each drop step (0.5 corresponds to halving), and r is the so-called **drop rate** defining the length of drop steps (10 corresponds to a drop every 10 epochs). The floor operator in the above formula equals 0 if the value of its argument is less than 1.

Exponential learning rate decay schedules are similar to step-based. However, instead of drop steps, a decreasing exponential function is used. The mathematical formula for the learning rate update, according to a popular exponential learning rate decay schedule, is:

$$\alpha_n \leftarrow \alpha_0 e^{-d \times n}$$

where d is the decay rate and e is **Euler's number**.

There are several popular upgrades to **minibatch SGD**, such as Momentum, Root Mean Squared Propagation (RMSProp), and Adam. These algorithms update the learning rate automatically based on the performance of the learning process. You don't have to worry about choosing the initial learning rate value, the decay schedule and rate, or other related hyperparameters. These algorithms have demonstrated good performance in practice, and practitioners often use them instead of manually tuning the learning rate.

Momentum helps accelerate minibatch SGD by orienting the gradient descent to the relevant direction, and reducing oscillations. Instead of using only the current gradient's epoch to guide the search, Momentum accumulates the gradient of past epochs to determine the direction to go. Momentum removes the need to manually adjust the learning rate.

More recent advancements in neural network cost function optimization algorithms include **RMSProp** and **Adam**, the latter being the most recent and versatile. It's recommended to start training the model with Adam. Then, if the quality of the model doesn't reach the acceptable level, try a different cost function optimization algorithm.

6.1.6 Regularization

In neural networks, besides **L1** and **L2 regularization**, you can use neural network-specific regularizers: dropout, early stopping, and batch-normalization. The latter is technically not a regularization technique, but it often has a regularization effect on the model.

The concept of **dropout** is very simple. Each time you "run" a training example through the network, you temporarily exclude at random some units from the computation. The higher the percentage of units excluded, the stronger the regularization effect. Popular neural network libraries allow you to add a dropout layer between two successive layers, or you can specify the dropout hyperparameter for a layer. The dropout hyperparameter varies in the range $[0, 1]$ and characterizes the fraction of units to randomly exclude from computation. The value of the hyperparameter has to be found experimentally. While simple, dropout's flexibility and regularizing effect are phenomenal.

Early stopping trains a neural network by saving the preliminary model after every epoch. Models saved after each epoch are called **checkpoints**. Then it assesses each checkpoint's performance on the validation set. You'll find during gradient descent that the cost decreases as the number of epochs increases. After some epoch, the model can start overfitting, and the model's performance on the validation data can deteriorate. Remember the bias-variance illustration in Figure **??** in Chapter 5. By keeping a version of the model after each epoch, you can stop the training once you start observing a decreased performance on the validation set. Alternatively, you can keep running the training process for a fixed number of epochs, and then pick the best checkpoint. Some machine learning practitioners rely on this technique. Others try to properly regularize the model using appropriate techniques.

Batch normalization (which rather should be called batch standardization) consists of **standardizing** the outputs of each layer before the next layer receives them as input. In practice, batch normalization results in faster and more stable training, as well as some regularization effect. So, it's always a good idea to use batch normalization. In popular neural network libraries, you can often insert a batch normalization layer between two subsequent layers.

Another regularization technique that can be applied to any learning algorithm is **data augmentation**. This technique is often used to regularize models that work with images. In practice, applying data augmentation often results in an increased model performance.

6.1.7 Network Size Search and Hyperparameter Tuning

Step 5 of the deep model training strategy is similar to that in the shallow model training strategy — choose a hyperparameter tuning strategy T.

It step 6, we pick a combination of hyperparameter values using strategy T. Typical parameters include the size of the minibatch, the value of the learning rate (if you use the vanilla minibatch SGD), or an algorithm that automatically updates the learning rate, such

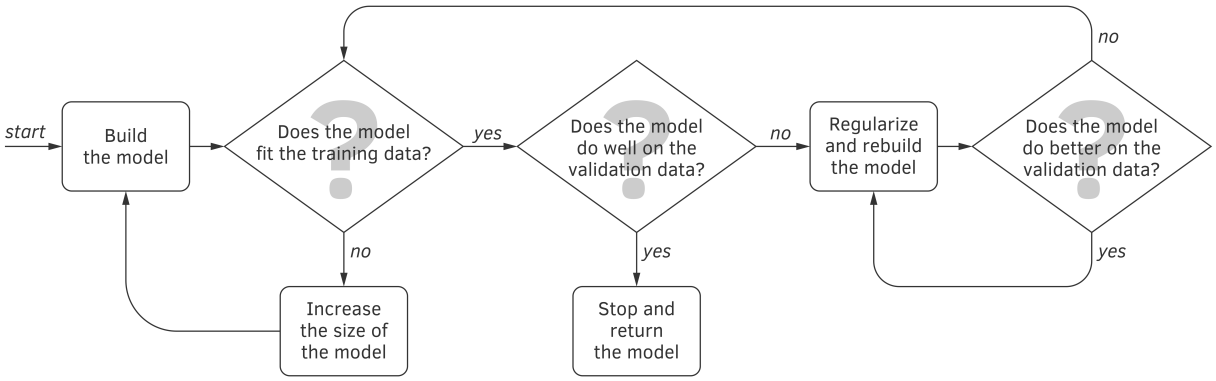


Figure 3: The neural network model training flowchart.

as Adam. You also decide the initial number of layers and units per layer. It's recommended to start with something reasonable that would allow us to build the first model fast enough. For example, two hidden layers and 128 units per layer might be a good starting point.

Step 7 reads, "Build the training model M, using algorithm A, parametrized with hyperparameters H, to optimize the cost function C." This is the main difference with shallow learning. When you work with a shallow learning algorithm or a model, you can only tweak some built-in hyperparameters. You don't have much control over the model architecture and complexity. With neural networks, you have all the control, and training a model is more a process than a single action. To build a deep model, you start with a reasonably-sized model, and then you follow the flowchart shown in Figure 3.

Observe that you start with some model, and then increase its size until it fits the training data well. Then you evaluate the model on the validation data. If it performs well, according to the performance metric, you stop and return the model. Otherwise, you regularize and retrain the model.

As we have seen, regularization in neural networks is usually achieved in several ways. The most effective is **dropout**, where you randomly remove some units from the network and make it simpler and "dumber." A simpler model would work better on the holdout data, and this is your goal.

Suppose, after several loops of regularization and model retrains, you don't see any improvement in the model performance on the validation data. Check if it still fits the training data. If it doesn't, increase the size of the model, by increasing the size of individual layers, or by adding another layer. Continue until the model fits the training data again. Then evaluate it again on the validation data. The process continues until a larger model doesn't result in better validation data performance, no matter your actions. Then you stop and return the model, if validation data performance is satisfactory.

If you are not satisfied with this performance, you can pick a different combination of hyperparameters for step 8, and build a different model. You will continue to test different values of hyperparameters until there are no more values to test. Then you keep the best model among those you trained in the process. If the performance of the best model is still not satisfactory, try a different network architecture, add more labeled data, or try **transfer learning**. We talk more on transfer learning in Section 6.1.10.

The properties of a trained neural network depend a lot on the choice of the values of hyperparameters. But before you choose specific values of hyperparameters, train a model, and validate its properties on the validation data, you must decide which hyperparameters are important enough for you to spend the time on.

Obviously, if you had infinite time and computing resources, you would tune all hyperparameters. However, in practice, you have finite time and, often, relatively modest resources. Which hyperparameters to tune?

While there is no definitive answer to that question, there are several observations that might help you in choosing the hyperparameters to tune when you work on a specific model:

- your model is more sensitive to some hyperparameters than to others; and
- the choice is often between using the default value of a hyperparameter or changing it.

The libraries for training neural networks often come with default values for hyperparameters: stochastic gradient descent version (often, **Adam**), the parameter initialization strategy (often, **random normal** or **random uniform**), minibatch size (often, 32), and so on. Those defaults were chosen based on observations from practical experience. Open-source libraries and modules are often the fruit of the collaboration of many scientists and engineers. These talented and experienced people established "good" defaults for many hyperparameters when working with various datasets and practical problems.

If you decide to tune a hyperparameter, as opposed to using the default value, it makes more sense to tune the hyperparameters to which the model is sensitive. Table 1 shows[2] several hyperparameters and approximate sensitivity of a neural network to those hyperparameters.

6.1.8 Handling Multiple Inputs

In practice, machine learning engineers often work with multimodal data. For example, the input could be an image and a text, and the binary output could indicate whether the text describes the given image.

It's hard to adapt **shallow learning** algorithms to work with multimodal data. For example, you can try to vectorize each input, by applying the corresponding feature engineering method. Then, concatenate two feature vectors to form one wider feature vector. If your image has features $[i^{(1)}, i^{(2)}, i^{(3)}]$, and your text has features $[t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}]$, your concatenated feature vector will be $[i^{(1)}, i^{(2)}, i^{(3)}, t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}]$.

[2]Taken from the talk "Troubleshooting Deep Neural Networks" by Josh Tobin et al., January 2019.

Hyperparameter	Sensitivity
Learning rate	High
Learning rate schedule	High
Loss function	High
Units per layer	High
Parameter initialization strategy	Medium
Number of layers	Medium
Layer properties	Medium
Degree of regularization	Medium
Choice of optimizer	Low
Optimizer properties	Low
Size of minibatch	Low
Choice of non-linearity	Low

Table 1: Approximate sensitivity of a model to some hyperparameters.

With neural networks, you have substantially more flexibility. You can build two **subnetworks**, one for each input type. For example, a **CNN** subnetwork reads the image, while an **RNN** subnetwork reads the text. Both subnetworks have, as their last layer, an **embedding**. CNN has an image embedding, and RNN has a text embedding. You then concatenate the two embeddings, and finally add a classification layer, such as **softmax** or **logistic sigmoid**, on top of the concatenated embeddings.

Neural network libraries provide simple-to-use tools that allow concatenating or averaging layers from several subnetworks.

6.1.9 Handling Multiple Outputs

Sometimes, you would like to predict multiple outputs for one input. Some problems with multiple outputs can be effectively converted into a multi-label classification problem. Those with labels of the same nature (like tags in social networks), or fake labels can be created as a full enumeration of combinations of original labels.

However, in many cases, the outputs are multimodal, and their combinations cannot be effectively enumerated. Consider the following example: you want to build a model that detects an object on an image, and returns its coordinates. In addition, the model has to return a tag describing the object, such as "person," "cat," or "hamster." Your training example will be a feature vector representing an image and a label. The label could be represented as a vector of coordinates of the object, and another vector with a **one-hot encoded** tag.

For this, you can create one subnetwork that works as an encoder. It will read the input image using, for example, one or several convolution layers. The encoder's last layer is the

image embedding. Then you add two other subnetworks on top of the embedding layer: 1) one takes the embedding vector as input, and predicts the coordinates of the object, and 2) the other takes the embedding vector as input, and predicts the tag.

The first subnetwork can have a **ReLU** as the last layer, which is good for predicting positive real numbers, such as coordinates. This subnetwork can use the mean squared error cost C_1. The second subnetwork will take the same embedding vector as input, and will predict the probabilities for each tag. It can have a **softmax** as the last layer, which is appropriate for the **multiclass classification**, and use the averaged **negative log-likelihood cost** C_2 (also called **cross-entropy** cost). Alternatively, the coordinates could be in the range $[0, 1]$ (in which case the layer that predicts coordinates will have four **logistic sigmoid** outputs and average four **binary cross-entropy** cost functions), while the layer that predicts tags might solve a **multi-label classification** problem (in which case it would also have several sigmoid outputs and average several binary cross entropy costs, one per tag).

Obviously, you are interested in accurate predictions of both the coordinates and the tags. However, it is impossible to optimize two cost functions at once. By trying to improve one, you risk hurting the other one, and vice-versa. What you can do is add another hyperparameter γ, in the range $(0, 1)$, and define the combined cost function as $\gamma \times C_1 + (1 - \gamma) \times C_2$. Then you tune the value for γ on the validation data, just like any other hyperparameter.

6.1.10 Transfer Learning

Recall, **transfer learning** consists of using a pre-trained model to build a new model. Pre-trained models are usually created using big data available to its creators, usually large organizations, but not necessarily available to you. The parameters learned by the pre-trained models can be useful for your task.

A pre-trained model can be used in two ways:

1) its learned parameters can be used to initialize your own model, or
2) it can be used as a feature extractor for your model.

Using Pre-Trained Model as Initializer

As discussed, the choice of parameter initialization strategy affects the properties of the learned model. Pre-trained models, whether available on the Internet, or trained by you, usually perform well for solving the original learning problem.

If your current problem is similar to the one solved by the pre-trained model, chances are high that the optimal parameters for your current problem will not be too different from the pre-trained parameters, especially in the initial neural network layers (those closest to the input).

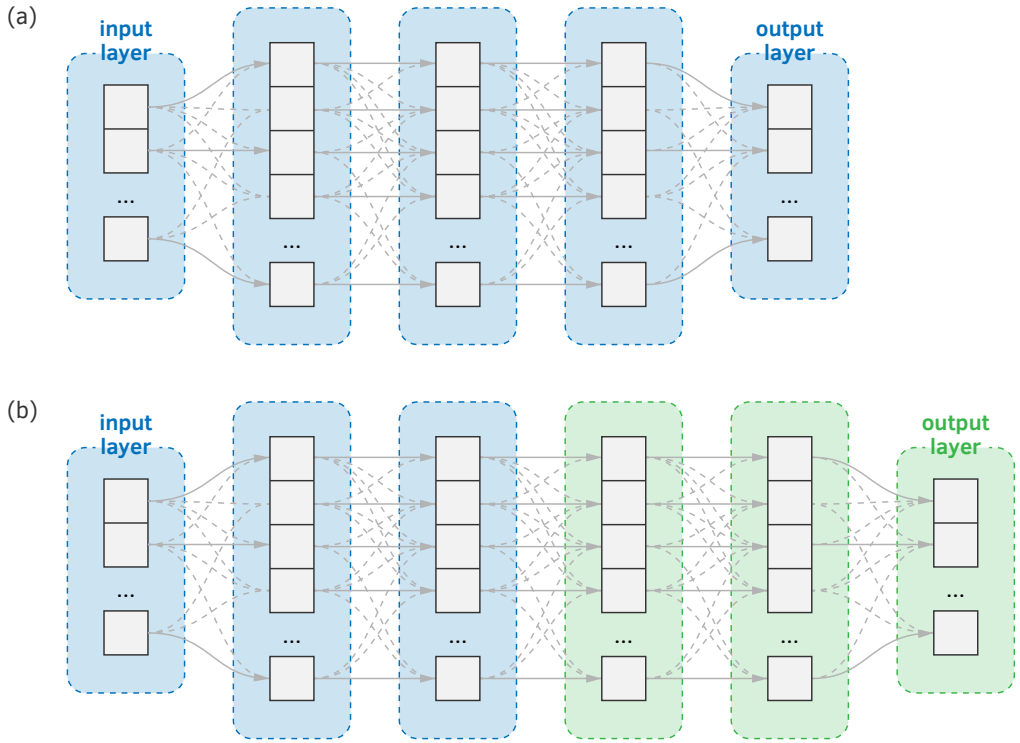


Figure 4: An illustration of transfer learning: (a) a pre-trained model and (b) your model, where you used the left part of the pre-trained model, and added new layers, including a different output layer tailored for your problem.

The learning might go faster for your problem because gradient descent will search for the optimal parameter values in a smaller region of potentially good values.

If the pre-trained model was built using a training set much bigger than yours, searching in a region of potentially good values might also lead to a better generalization. Indeed, if some behavior of the model you want to build is not reflected in your training examples, this behavior could still be "inherited" from the pre-trained model.

Using Pre-Trained Model as Feature Extractor

If you use a pre-trained model as an initializer for your model, it gives you more flexibility. The gradient descent will modify the parameters in all layers, and, potentially, reach a better performance for your problem. The downside of that is you will often end up training a very deep neural network.

Some pre-trained models contain hundreds of layers and millions of parameters. Training a large network like that can be challenging. It will definitely require a significant amount of computational resources. In addition, the problem of the vanishing gradient is more severe in a deep neural network than one with a couple hidden layers.

If you have a limited amount of computational resources, you might prefer using some layers of the pre-trained model as **feature extractors** for your model. In practice, it means that you only keep several initial layers of the pre-trained model, those closest to and including the input layer. You keep their parameters "frozen," that is, unchanged and unchangeable. Then you add new layers on top of the frozen layers, including the output layer appropriate for your task. Only the parameters of the new layers will be updated by gradient descent during training on your data.

An illustration of the process is shown in Figure 4. The blue neural network is a pre-trained model. Some of the blue layers are reused in the new model with their parameters frozen; the green layers are added by the analyst and tailored to the problem at hand.

The analyst might decide to freeze the parameters of the entire blue part of the new network, and only train the parameters of the green part. Alternatively, several right-most blue layers could be set as trainable.

How many layers of the pre-trained model to use in the new model? Freeze how many layers? This is up to the analyst: it's part of the decisions you'll make about the architecture that will work best for your problem.

6.2 Stacking Models

Ensemble learning is training an ensemble model, which is a combination of several **base models**, each individually performing worse than the ensemble model.

6.2.1 Types of Ensemble Learning

There are ensemble learning algorithms, such as **random forest learning** and gradient boosting. They train an ensemble of several hundred to thousands of **weak models**, and obtain a **strong model** that has a significantly better performance than the performance of each weak model. We will not discuss these algorithms here. If you are missing this knowledge, it can easily be found in a specialized machine learning book.[3]

The reason why combining multiple models can bring better performance is that, when several uncorrelated models agree, they are more likely to agree on the correct outcome. The key word here is "uncorrelated." Ideally, base models should be obtained by using different features, or be of a different nature — for example, SVM and random forest. Combining

[3]You can read about ensemble learning algorithms in Chapter 7 of The Hundred-Page Machine Learning Book.

different versions of the decision tree learning algorithm, or several SVMs with different hyperparameters, may not result in a significant performance boost.

The goal of ensemble learning is to learn to combine the strengths of each base model. There are three ways to combine weakly correlated models into an ensemble model: 1) averaging, 2) majority vote, and 3) model stacking.

Averaging works for regression, as well as those classification models that return classification scores. It consists of applying all your base models to the input $\mathbf{x}$, and then averaging the predictions. To see if the averaged model works better than each individual algorithm, you can test it on the validation set using a metric of your choice.

Majority vote works for classification models. It consists of applying all your base models to the input $\mathbf{x}$, and then returning the majority class among all predictions. In the case of a tie, you can either randomly pick one of the classes, or return an error message if misclassifying would incur a significant loss for the business.

Model stacking is an ensemble learning method that trains a strong model by inputting the outputs of other strong models. Let's go into more detail about model stacking.

6.2.2 An Algorithm of Model Stacking

Say you want to combine classifiers f_1, f_2, and f_3, all predicting the same set of classes. To create a synthetic training example $(\hat{\mathbf{x}}_i, \hat{y_i})$ for the stacked model from the original training example $(\mathbf{x}_i, y_i)$, set $\hat{\mathbf{x}}_i \leftarrow [f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x})]$, and $\hat{y}_i \leftarrow y_i$. This is illustrated in Figure 5.

If some of your base models return a class plus a class score, you can use those scores as additional input features for the stacked model.

To train the stacked model, use synthetic examples, and tune the hyperparameters of the stacked model using cross-validation. Make sure your stacked model performs better on the validation set than each of the stacked base models.

In addition to using different machine learning algorithms and models, some base models, to be weakly correlated, can be trained by randomly sampling the examples and features of the original training set. Furthermore, the same learning algorithm, trained with very different hyperparameter values, could produce sufficiently uncorrelated models.

6.2.3 Data Leakage in Model Stacking

To avoid **data leakage**, be careful when training a stacked model. To create the synthetic training set for the stacked model, follow a process similar to cross-validation. First, split all training data into ten or more blocks. The more blocks the better, but the process of training the model will be slower.

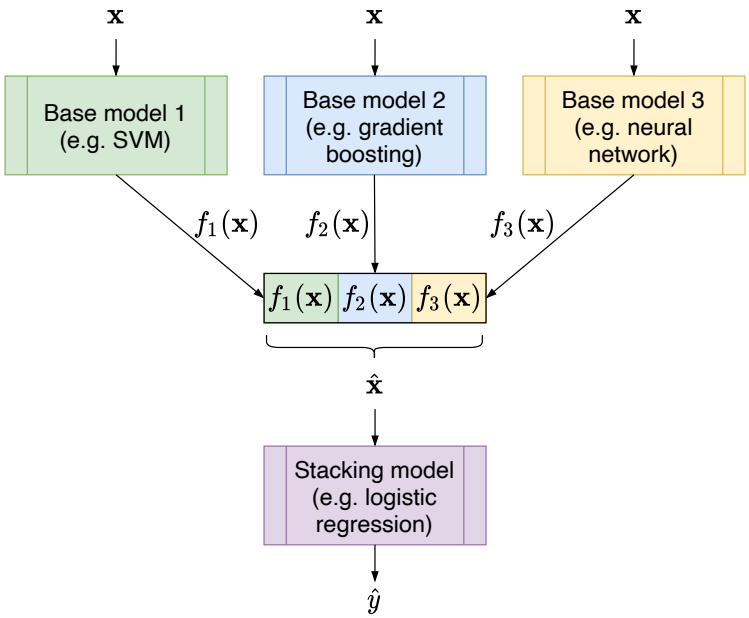


Figure 5: A stacking of three weakly correlated strong models.

Temporarily exclude one block from the training data, and train the base models on the remaining blocks. Then apply the base models to the examples in the excluded block. Obtain the predictions, and build the synthetic training examples for the excluded block by using the predictions from the base models.

Repeat the same process for each of the remaining blocks, and you will end up with the training set for the stacking model. The new synthetic training set will be of the same size as that of the original training set.

6.3 Dealing With Distribution Shift

Recall that the holdout data must resemble the data you will observe in production. Sometimes, however, it is not available in sufficiently large quantities. At the same time, you might have access to labeled data that is similar to the production data, but not exactly the same. For example, you might have lots of labeled images from the Web crawl collection, but your goal is to train a classifier for Instagram photos. You might not have enough labeled Instagram photos for training, so you hope to train the model by using the Web crawl data, and then be able to use that model to classify the Instagram photos.

6.3.1 Types of Distribution Shift

When the distributions of the training data and test data are not the same, we call it **distribution shift**. Dealing with a distribution shift is currently an open research area. Researchers distinguish three types of distribution shift:

- **covariate shift** — shift in the values of features;
- **prior probability shift** — shift in the values of the target; and
- **concept drift** — shift in the relationship between the features and the label.

You may know your data is affected by a distribution shift, but you don't usually know what type of shift it is.

If the number of examples in the test set is relatively high compared to the size of the training set, you could randomly pick a certain fraction of test examples and transfer some to the training set and some to the validation set. Then you would train the model as usual. However, often you have a very high number of training examples and relatively few test examples. In that case, a more effective approach is to use **adversarial validation**.

6.3.2 Adversarial Validation

We prepare for adversarial validation as follows. We assume that the feature vectors in a training and a test examples contain the same number of features, and those features represent the same information. Split your original training set into two subsets: Training Set 1 and Training Set 2.

Create a Modified Training Set 1 by transforming the examples from Training Set 1 as follows. To each example in Training Set 1, add the original label as an additional feature, then assign the new label "Training" to that example.

Create a Modified Test Set by transforming the examples from the original test set as follows. To each example in the test set, add the original label as an additional feature, then assign the new label "Test" to that example.

Merge the Modified Training Set 1 and the Modified Test Set to obtain a new Synthetic Training Set. You will use it for solving a binary classification problem of distinguishing the "Training" examples from the "Test" examples. Use that Synthetic Training Set, and train a binary classifier that returns a prediction score.

Observe that the binary classifier we have trained will predict, for a given original example, whether it's a training or a test example. Apply that binary classifier to the examples from Training Set 2. Identify the examples predicted as "Test," which the binary model is most certain about. Use those examples as validation data for your original problem.

Remove the examples from Training Set 1 which the binary model predicted "Training" with the highest certainty. Use the remaining examples in Training Set 1 as the training data for your original problem.

You must experiment to find out what is the ideal way to split the original training set into Training Set 1 and Training Set 2. You also must find out how many examples from Training Set 1 to use for training, and how many of them to use for validation.

6.4 Handling Imbalanced Datasets

In Section **??** of Chapter 3, we considered some techniques to handle **imbalanced datasets**, such as over- and undersampling, and generating synthetic data.

In this section, we will consider additional techniques that are applied during learning, as opposed to in the data collection and preparation stage.

6.4.1 Class Weighting

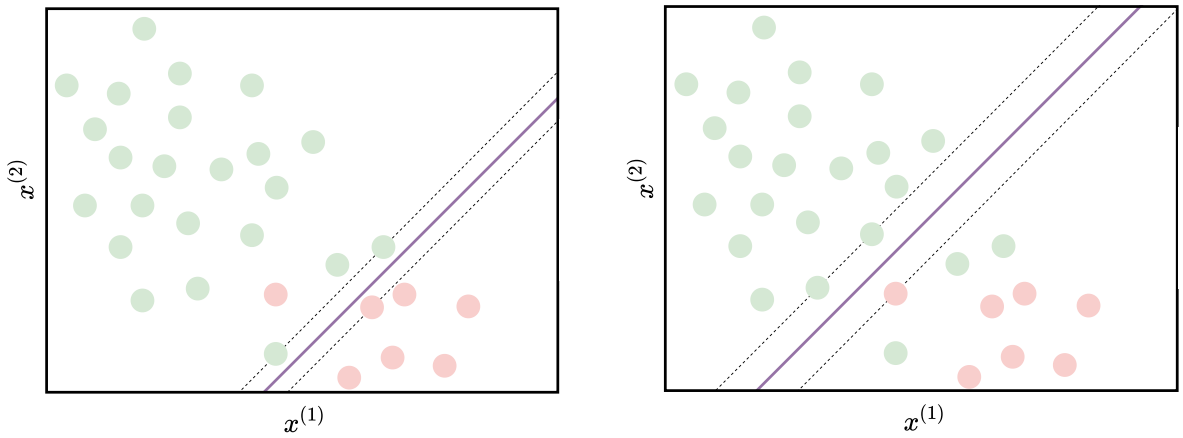
Some algorithms and models, such as **support vector machine** (SVM), **decision trees**, and **random forests**, allow the data analyst to provide weights for each class. The loss in the cost function is typically multiplied by the weight. The data analyst may, for example, provide greater weight to the minority class. This makes it harder for the learning algorithm to disregard examples of the minority class, because it would result in much higher cost than without class weighting.

Let's see how it works in support vector machines. Our problem is distinguishing between genuine and fraudulent e-commerce transactions. The examples of genuine transactions are much more frequent. If you use SVM with **soft margin**, you can define a cost for misclassified examples. The SVM algorithm tries to move the hyperplane to reduce the number of misclassified examples. If the misclassification cost is the same for both classes, the "fraudulent" examples, in the minority, risk being misclassified to allow classifying more of the majority class correctly. This situation is illustrated in Figure 6a. This problem is observed for most learning algorithms applied to imbalanced datasets.

If you set higher the loss of minority misclassification, then the model will try harder to avoid misclassifying those examples. But this will incur the cost of misclassification of some majority class examples, as illustrated in Figure 6b.

6.4.2 Ensemble of Resampled Datasets

Ensemble learning is another way of mitigating the class imbalance problem. The analyst randomly chunks majority examples into H subsets, then creates H training sets. After training H models, the analyst then makes predictions by averaging (or taking the majority) of the outputs of H models.



(a) (b)

Figure 6: An illustration of an imbalanced problem. (a) Both classes have the same weight; (b) examples of the minority class have a higher weight.

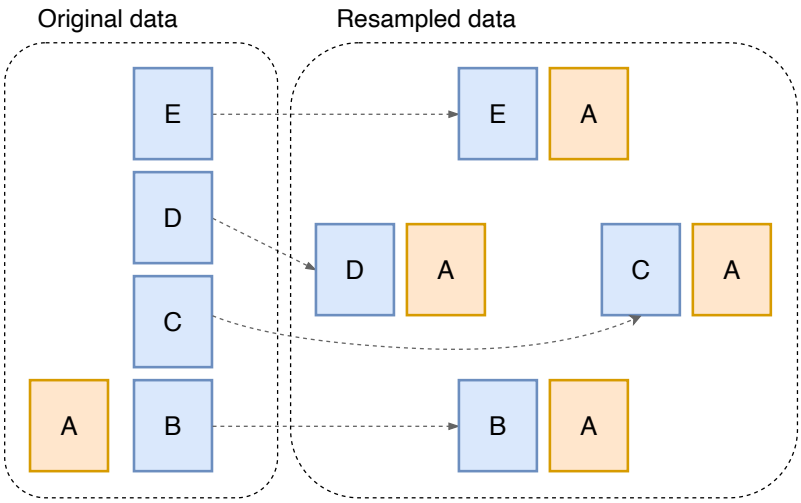


Figure 7: An ensemble of resampled datasets.

The process for $H = 4$ is illustrated in Figure 7. Here, we transformed our imbalanced binary learning problem into four balanced problems by chunking the examples of the majority class into four subsets. The examples of the minority class are copied four times in their entirety.

This approach is simple and scalable: you can train and run your models on different CPU cores or cluster nodes. Ensemble models also tend to produce a better prediction than each individual model in the ensemble.

6.4.3 Other Techniques

If you use stochastic gradient descent, the class imbalance can be tackled in several ways. First, you can have different learning rates for different classes: a lower value for the examples of the majority class, and a higher value otherwise. Second, you can make several consecutive updates of the model parameters each time you encounter an example of a minority class.

For imbalanced learning problems, the performance of the model is measured using adapted performance metrics such as **per-class accuracy** and **Cohen's kappa statistic** that we considered in Section **??** in the previous chapter.

6.5 Model Calibration

Sometimes it is important that the classification model returns not just the predicted class, but also the probability that the predicted class is correct. Some models return a score along with the predicted class. Even if its value ranges between 0 and 1, it's not always a probability.

6.5.1 Well-Calibrated Models

We say that the model is **well-calibrated** when, for input example $\mathbf{x}$ and predicted label $\hat{y}$, it returns the score that can be interpreted as the true probability for $\mathbf{x}$ to belong to class $\hat{y}$.

For instance, a well-calibrated binary classifier would generate a score of 0.8 for approximately 80% of the examples actually belonging to the positive class.

Most machine learning algorithms train models that are not well-calibrated, as shown[4] by the **calibration plots** in Figure 8.

A calibration plot for a binary model allows seeing how well the model is calibrated. On the X-axis, there are bins that group examples by the predicted score. For example, if we have 10 bins, the left-most bin groups all examples for which the predicted score is in the range $[0, 0.1)$ while the right-most bin groups all examples for which the predicted score is in the range $[0.9, 1.0]$. On the Y-axis, there are the fractions of positive examples in each bin.

[4]The graph is adapted from https://scikit-learn.org/stable/modules/calibration.html.

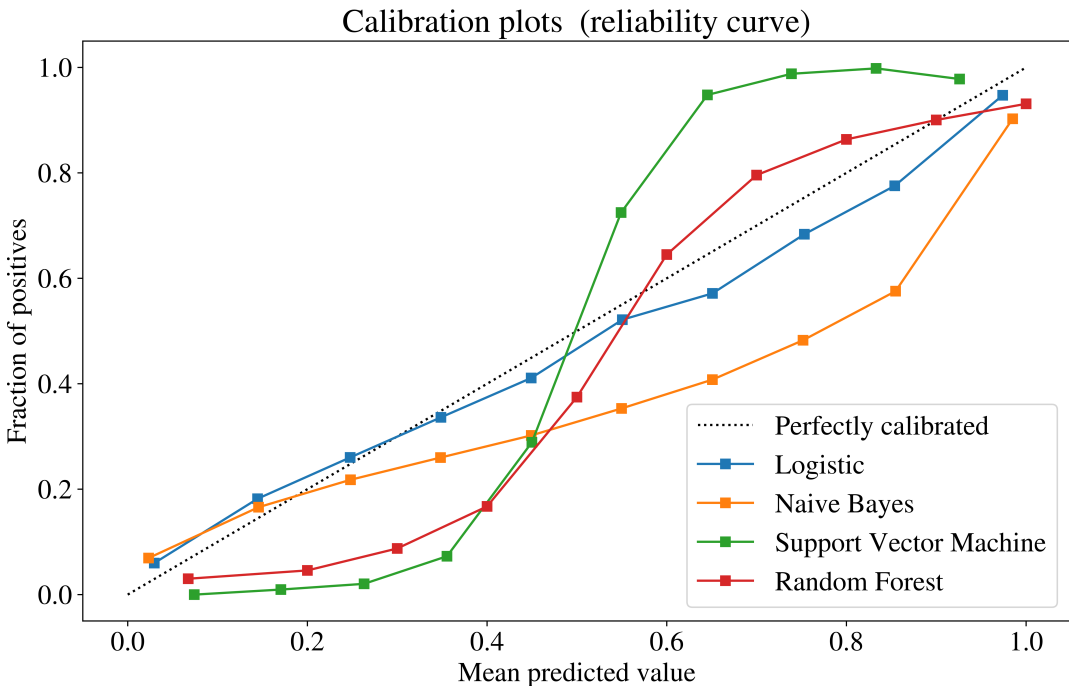


Figure 8: Calibration plots for models trained by several machine learning algorithms applied to a random binary dataset.

For multiclass classification, we would have one calibration plot per class in a **one-versus-rest** way. One-versus-rest is common strategy for converting a binary classification learning algorithm for solving multiclass classification problems. The idea is to transform a multiclass problem into C binary classification problems and build C binary classifiers. For example, if we have three classes, $y \in \{1, 2, 3\}$, we create three original dataset copies, and modify them. In the first copy, we replace all labels not equal to 1 with a 0. In the second copy, we replace all labels not equal to 2 with a 0. In the third copy, we replace all labels not equal to 3 with a 0. Now we have three binary classification problems where we want to learn to distinguish between labels 1 and 0, 2 and 0, and 3 and 0. As you can see, in each of the three binary classification problems, the label 0 denotes the "rest" in "one-versus-rest."

When the model is well-calibrated, the calibration plot oscillates around the diagonal (shown as a dotted line in Figure 8). The closer the calibration plot is to the diagonal, the better the model is calibrated. Because a logistic regression model returns the true probabilities of the positive class, its calibration plot is closest to the diagonal. When the model is not well-calibrated, the calibration plot usually has a sigmoid-shape, as shown by the **support vector machine** and **random forest** models.

6.5.2 Calibration Techniques

There are two techniques often used to calibrate a binary model: **Platt scaling** and **isotonic regression**. The two are based on similar principles.

Let us have a model f that we want to calibrate. First of all, we need a holdout dataset specifically set aside for calibration. To avoid overfitting, we cannot use training or validation data for calibration. Let this calibration dataset be of size M. Then, we apply the model f to each example $i = 1, \dots, M$ and obtain, for each example i, the prediction f_i. We build a new dataset $\mathcal{Z}$, where each example is a pair (f_i, y_i), y_i is the true label of example i, and labels have the values in the set $\{0, 1\}$.

The only difference between Platt scaling and isotonic regression is that the former builds a logistic regression model by using the dataset $\mathcal{Z}$, while the latter builds the isotonic regression of $\mathcal{Z}$, that is, a non-decreasing function as close to the examples as possible. Once we have the calibration model z, obtained either using Platt scaling or isotonic regression, we can predict the calibrated probability for an input $\mathbf{x}$ as $z(f(\mathbf{x}))$.

Notice that a calibrated model may or may not result in better quality prediction for your problem. That depends on the chosen model performance metric.

According to experiments:[5] Platt scaling is most effective when the distortion in the predicted probabilities is sigmoid-shaped. Isotonic regression can correct a wider range of distortions. Unfortunately, this extra power comes at a price. Analysis has shown that isotonic regression is more prone to overfitting, and thus performs worse than Platt scaling when data is scarce.

Experiments with eight classification problems also suggested that random forests, neural networks, and bagged decision trees are the best learning methods for predicting well-calibrated probabilities prior to calibration, but after calibration, the best methods are boosted trees, random forest, and SVM.

6.6 Troubleshooting and Error Analysis

Troubleshooting a machine learning pipeline is hard. It's difficult to differentiate whether the model performs poorly because your code contains a bug, or if there are problems with your training data, learning algorithm, or the way you designed your pipeline. Moreover, the same degradation in performance can be explained by various reasons. The results of the learning can be sensitive to small changes in hyperparameters or dataset makeup.

Because of these challenges, model training is usually an iterative process, where an analyst trains a model, observes its behavior, and makes adjustments based on observations.

[5] Alexandru Niculescu-Mizil and Rich Caruana, "Predicting Good Probabilities With Supervised Learning", appearing in Proceedings of the 22nd International Conference on Machine Learning, Bonn, Germany, 2005.

6.6.1 Reasons for Poor Model Behavior

If your model does poorly on the training data (underfits it), common reasons are:

- the model architecture or learning algorithm are not expressive enough (try more advanced learning algorithm, an **ensemble method**, or a deeper **neural network**);
- you regularize too much (reduce **regularization**);
- you have chosen suboptimal values for hyperparameters (**tune hyperparameters**);
- the features you engineered don't have enough **predictive power** (add more informative features);
- you don't have enough data for the model to generalize (try to get more data, use **data augmentation**, or **transfer learning**); or
- you have a bug in your code (debug the code that defines and trains the model).

If your model does well on the training data, but poorly on the holdout data (overfits the training data), common reasons are:

- you don't have enough data for generalization (add more data or use data augmentation);
- your model is under-regularized (add regularization or, for neural networks, both regularization and **batch normalization**);
- your training data distribution is different from the holdout data distribution (reduce the **distribution shift**);
- you have chosen suboptimal values for hyperparameters (tune hyperparameters); or
- your features have low predictive power (add features with high predictive power).

6.6.2 Iterative Model Refinement

If you have access to new labeled data (for example, you can label examples yourself, or easily request the help of a labeler) then, you can refine the model using a simple iterative process:

1. Train the model using the best values of hyperparameters identified so far.
2. Test the model by applying it to a small subset of the validation set ($100-300$ examples).
3. Find the most frequent error patterns on that small validation set. Remove those examples from the validation set, because your model will now overfit to them.
4. Generate new features, or add more training data to fix the observed error patterns.
5. Repeat until no frequent error patterns are observed (most errors look dissimilar).

Iterative model refinement is a simplified version of **error analysis**. A more principled approach is described below.

6.6.3 Error Analysis

Errors can be:

- uniform, and appear with the same rate in all use cases, or

- focused, and appear more frequently in certain types of use cases.

Focused errors following a specific pattern are those that merit special attention. By fixing an error pattern, you fix it once for many examples. Focused errors, or error trends, usually happen when some use cases aren't well-represented in the training data. For example, a face detection system developed by a major web camera provider worked better for white users than for black users. In another case, a human presence detection system equipped with a night vision system worked better during the day than at night, simply because the night training examples were less frequent in the training data.

Uniform errors cannot be entirely avoided, but important focused errors should be discovered before the model is deployed in production. This can be done by clustering test examples, and by testing the model on examples coming from different clusters. The distribution of the production (online) data can be significantly different from the offline data distribution used for model training/pre-deployment tests. So, the clusters that contain few examples in the offline data might represent much more frequent use cases in the online scenario.

In Section **??** of Chapter 4, we discussed several techniques for dimensionality reduction. In addition to using clustering for spotting error trends, uniform manifold approximation and projection (**UMAP**) or **autoencoder** can be used. Use those techniques to reduce the dimensionality of the data to 2D, and then visually inspect the distribution of errors across a dataset.

More specifically, you can visualize the data on a 2D scatter plot, using different colors for examples of different classes. To identify error trends on a scatter plot, use different markers depending on whether a model's prediction was correct or not. For example, use circles to denote examples whose label was predicted correctly, and squares otherwise. This will allow you to see the regions of poor model performance. If you work with perceptive data, such as images or text, it is also helpful to visually examine some examples from those poor performance regions.

Whether you are satisfied or dissatisfied by the model's performance on the holdout data, you can always improve the model by analyzing individual errors. As discussed, the best way is to work iteratively, by considering $100 - 300$ examples at a time. By considering a small number of examples at a time, you can iterate quickly, by retraining the model after each iteration, but still consider enough examples to spot obvious patterns.

How do you decide whether an error pattern is worth spending time to fix it? You can base that decision on the **error pattern frequencies**. Let's see how it works.

Let your model have an accuracy of 80%, which corresponds to an error rate of 20%. If you fix all error patterns, you can improve the model's performance by at most 20 percentage points. If your small error-analysis batch was of 300 examples, your model made $0.2 \times 300 = 60$ errors.

Observe the errors one by one, and try to get an idea of what particularities in the input led to a misclassification of those 60 examples. To be even more concrete, let our classification problem be to detect pedestrians-on-the-street images. Assume that in 60 out of the 300 images, the model failed to detect a pedestrian. After closer analysis, you discover two

patterns: 1) the image is blurry in 40 examples, and 2) the picture was taken during the nighttime in 5 examples. Now, should you spend time addressing both problems?

If you address the blurry-image problem (for example, by adding more labeled blurry images to your training data), you can hope to decrease your error by $(40/60) \times 20 = 13$ percentage points. In the best-case scenario, after you solve the blurry-image misclassification problem, your error becomes $20 - 13 = 7$ percent, a significant decrease from the initial 20% error.

On the other hand, if you solve the nighttime image problem, you can hope to decrease your error by $5/60 \times 20 = 1.7$ percentage points. So, in the best-case scenario, your model will make $20 - 1.7 = 18.3$ percent errors, which might be significant for some problems, or insignificant for others. The cost of gathering additional labeled night-time images can be significant and might not be worth the effort.

To fix an error pattern, you can use one or a combination of techniques:

- preprocessing the input (e.g. image background removal, text spelling correction);
- data augmentation (e.g., blurring or cropping of images);
- labeling more training examples; and
- engineering new features that would allow the learning algorithm to distinguish between "hard" cases.

6.6.4 Error Analysis in Complex Systems

Let's say you work on a complex document classification system that consists of three chained models as shown below:

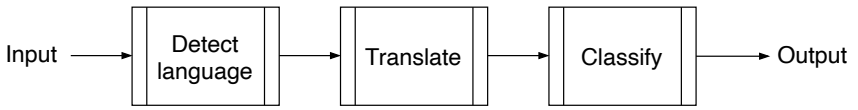


Figure 9: A complex document classification system.

Let the accuracy of the entire system be 73%. If the classification is binary, the accuracy of 73% doesn't seem high. On the other hand, if the classification model (the rightmost block in Figure 9) supports thousands of classes, then the accuracy of 73% doesn't seem too low. For some business cases, however, the user might expect human-like, or even superhuman performance.

Imagine that you are in a position, where the business expects a higher than 73% performance from the document classification system you have built. To get the most out of your additional effort, you must decide which part of the system needs improvement in the first place.

When the decision about something is made on several chained levels, like in the problem shown in Figure 9, and when those decisions are independent of one another, the accuracy multiplies.

For example, if the language predictor accuracy was 95%, the machine translation model accuracy[6] was 90%, and the classifier accuracy was 85%, then, in the case of independence of the three models, the overall accuracy of the entire three-stage system would be $0.95 \times 0.90 \times 0.85 = 0.73$, or 73 percent. At first glance, it seems obvious that the most gain in the entire system's accuracy would come from maximizing the accuracy of the third model — the classifier. However, in practice, some errors made by a given model might not significantly affect the overall performance of the system. For example, if the language predictor often confuses Spanish and Portuguese, the machine translation model could still be capable of generating an adequate translation for the third-stage classification model.

While working on the third-stage classifier, you might have concluded that you reached its maximum performance, so it doesn't make sense to continue. Now, which of the previous two models, the language detector and/or the machine translator, should you improve to increase the quality of the entire three-stage system?

One way to determine the upper bound of an entire system's potential is to perform the **error analysis by parts**. You replace one model's predictions with perfect labels, such as human-provided labels. Then you calculate how the entire system performs. For example, instead of using the machine translation system at stage two in Figure 9, you can ask a professional human translator to translate the text from the predicted language (if the prediction of the language was correct), or keep the original text (if the prediction of the language was wrong).

Let's say you asked a professional for a hundred translations. Now you can measure how perfect translations affect the overall system performance. Let the accuracy of the entire system's output become 74%. So, the potential gain from improved translation in overall system performance is only one percentage point. Reaching the human-level performance for a machine translation model can turn out to be a daunting task, not worth the effort, especially when what we can achieve in the end is one percentage point gain for the entire system. So, you might prefer spending more time on building a better language predictor in stage 1, if the potential gain in overall system performance prediction quality is higher.

6.6.5 Using Sliced Metrics

If the model will be applied to different segments of the use cases, it should be separately tested for each segment. For example, if you want to predict the solvency of borrowers, you would want your model to be equally accurate for both male and female borrowers. To achieve that, you can split your validation data into several subsets, one subset per segment. Then compute the performance metric by separately applying your model to each subset.

Alternatively, you can separately evaluate the model on each class by applying precision and recall metrics. Remember these metrics are defined only for binary classification. By isolating

[6]Measuring the error of the machine translation system in practice is tricky as a translation is rarely entirely accurate or inaccurate. Instead, measures, such as BLEU (for Bilingual Evaluation Understudy Score) score, are used.

one class in your multiclass classification problem, and labeling the other classes "Other," you can individually compute the precision and recall for each class.

If you see that the value of the performance metric changes between segments or classes, you can try to fix the problem by adding more labeled data to the segments or classes, where the performance of the model is unsatisfactory, or engineer additional features.

6.6.6 Fixing Wrong Labels

When humans label the training examples, the assigned labels can be wrong. This can cause poor model performance on the model on both training and holdout data. Indeed, if similar examples have conflicting labels — some correct and some incorrect — the learning algorithm can learn to predict the wrong label.

Here is a simple way to identify the examples that have wrong labels. Apply the model to the training data from which it was built, and analyze the examples for which it made a different prediction as compared to the labels provided by humans. If you see that some predictions are indeed correct, change those labels.

If you have time and resources, you could also examine the predictions with the score close to the decision threshold. Those are often mislabeled cases too.

If wrong labels in the training data is a serious issue, you can avoid it by asking several individuals to provide labels for the same training example. Only accept it if all individuals assigned the same label to that example. In less demanding situations, you can accept a label if the majority of individuals assigned it.

6.6.7 Finding Additional Examples to Label

As discussed above, error analysis can reveal that more labeled data is needed from specific regions of feature space. You might have an abundance of unlabeled examples. How should you decide which examples to label so as to maximize the positive impact on the model?

If your model returns a prediction score, an effective way is to use your best model to score the unlabeled examples. Then label those examples, whose prediction score is close to the prediction threshold.

When the error analysis has revealed error patterns by means of visualization, then choose those examples which are surrounded by many examples with prediction errors.

6.6.8 Troubleshooting Deep Learning

To avoid problems when training a deep model, follow a workflow shown below:

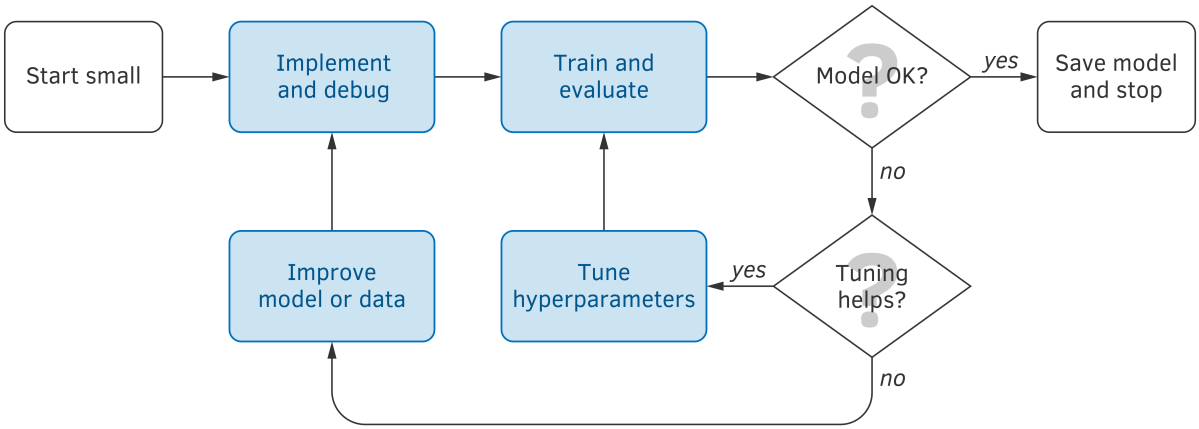


Figure 10: A deep learning troubleshooting workflow.

When possible, start small, for example, with a simple model using a high-level library, such as **Keras**. It should be very easy to validate visually, ideally fitting on at most two screens.

Alternatively, reuse an existing open-source architecture that was proven to work (pay attention to the code license!). Start with:

- a small, normalized dataset fitting in memory,
- the most simple to use cost-function optimizer (e.g., **Adam**),
- an initialization strategy (e.g., **random normal**),
- the default values of the sensitive hyperparameters of both the cost-function optimizer and the layers, and
- no regularization.

Once you have your first simplistic model architecture and dataset, temporarily reduce your training dataset even further, to the size of one **minibatch**. Then start the training. Make sure your simplistic model is capable of **overfitting** this training minibatch. If the overfitting of the minibatch doesn't happen, it is a solid indicator that something is wrong with your code or data. Look for the following signs[7] and their probable causes:

[7] Adapted from the talk "Troubleshooting Deep Neural Networks" by Josh Tobin et al., January 2019.

Sign	Probable causes
Error goes up	Flipped the sign of the loss function or gradient Learning rate too high Softmax taken over wrong dimension
Error explodes	Numerical issue Learning rate too high
Error oscillates	Data or labels corrupted (e.g. zeroed or incorrectly shuffled) Learning rate too high
Error plateaus	Learning rate too low Gradient not flowing through the whole model Too much regularization Incorrect input to loss function Data or labels corrupted

Table 2: Common issues and most common causes of problems with getting to overfit one minibatch by a neural network model.

Once your model overfits one minibatch, get back to the entire dataset, and train, evaluate, then tune hyperparameters until no improvements on the validation data are possible.

If the performance of the model is still unsatisfactory, update the model (e.g., by increasing its depth or width), or the training data (e.g., by changing the pre-processing, or adding features). Debug the change by overfitting one minibatch once again, then train, evaluate, and tune the new model. Keep iterating until you're satisfied with the quality of the model.

While you are searching for the best architecture for your model, it's convenient not just to use a smaller training set, but also to simplify the problem by either,

- creating a simple synthetic training set, or
- reducing the number of classes or the resolution of input images (or video fragments), size of the texts, bitrate of the sound frequencies, and so on.

At the evaluation step of the deep learning troubleshooting workflow shown in Figure 10, verify if the poor model performance could be caused by one of the reasons listed in Section 6.6.1. Choose the next step depending on whether the performance can be improved by tuning hyperparameters, updating the model, features, or the training data.

6.7 Best Practices

In this section, I gathered practical advice on training machine learning models. The best practices below aren't strict prescriptions. They are rather recommendations that often save time, effort, and might lead to higher quality results.

6.7.1 Deliver a Good Model

What is a good model? A good model has two properties:

- it has the desired quality according to the performance metric; and
- it is safe to serve in a production environment.

For a model to be safe-to-serve means satisfying the following requirements:

- it will not crash or cause errors in the serving system when being loaded, or when loaded with bad or unexpected inputs;
- it will not use an unreasonable amount of resources (such as CPU, GPU, or RAM).

6.7.2 Trust Popular Open Source Implementations

Modern open-source libraries and modules for machine learning in popular modern programming languages and platforms, such as Python, Java, and .NET, contain efficient, industry-standard implementations of popular machine learning algorithms. They usually have permissive licenses. Additionally, open-source libraries and modules exist specifically for training neural networks.

It is only considered reasonable to create your own machine learning algorithms if you use an exotic or very new programming language. In addition, you might program from scratch if the model is intended to be executed in a very resource-constrained environment, or you need to run your model with a speed no existing implementation can provide.

Avoid using multiple programming languages in the same project. Using different programming languages increases the cost of testing, deployment, and maintenance. It also makes it difficult to transfer project ownership between employees.

6.7.3 Optimize a Business-Specific Performance Measure

Learning algorithms try to reduce training data error. The data analyst, in turn, wants to minimize test data error. However, your client or employer typically wants you to optimize a **business-specific performance metric**.

When you have minimized the validation error rate, focus on tuning hyperparameters that optimize a business-specific metric, even if it causes the validation error rate to increase.

6.7.4 Upgrade From Scratch

Once deployed to production, some models have to be periodically updated with new data to adapt to the user's needs. This new training data must be automatically collected by using scripts (as we discussed in Chapter 3 in Section **??** about **reproducibility**).

Each time the data is updated, the hyperparameters must be tuned from scratch. Otherwise, the new data may yield suboptimal performance with old hyperparameters.

Some models, such as neural networks, may be iteratively upgraded. However, avoid the practice of **warm-starting**. It consists of iteratively upgrading the existing model by using only new training examples and running additional training iterations.

Furthermore, frequent model upgrades without retraining from scratch can lead to **catastrophic forgetting**. It's a situation in which the model that was once capable of something, "forgets" that capability because of learning something new.

Note that upgrading the model is not the same as **transfer learning**. Analysts use transfer learning when the data used to build the pre-trained model, or adequate computing resources, are not available.

6.7.5 Avoid Correction Cascades

You might have model m_A that solves problem A, but you need a solution m_B for a slightly different problem B. It can be tempting to use the output of m_A as input for m_B, and only train m_B on a small sample of examples that "correct" the output of m_A for solving problem B. Such technique is called **correction cascading**, and it is not recommended.

Model cascading makes it impossible to update model m_A, without also updating model m_B (and the rest of the cascade). The effect a change in m_A might have on m_B is impossible to predict, but most likely it will be negative. Furthermore, the developer of model m_B might not know about the change in model m_A, and the developer of model m_A might not know that model m_B depends on it. The negative effect on m_B of the change in model m_A may go unnoticed for a long time.

Instead of building a correction cascade, it is recommended to update model m_A to include the use cases for solving problem B. It would be wise to add features allowing the model to distinguish between the examples of problem B. One might also use transfer learning, or build an entirely independent model for solving problem B.

6.7.6 Use Model Cascading With Caution

It's important to note that **model cascading** is not always a bad practice. Using the output of one model, as one of many inputs for another model, is common. It might significantly reduce time to market. However, cascading must be used with caution, because the update of one model in a cascade must involve an update of all models in the cascade, which can end up being costly in the long-term.

To mitigate the negative effect of model cascading, two strategies are beneficial:

1. Analyze the information flow in your software system and update, or retrain, the entire chain. Model m_A's updated output must be reflected in the training data for model m_B.

2. Control who can and who cannot make calls to model m_A to prevent undeclared consumers from creating this issue. As Google's engineers mentioned:[8] "In the absence of barriers, engineers will naturally use the most convenient signal at hand, especially when working against deadline pressures."

Furthermore, a prediction output by a model should not be a plain number or a string. It should come with information about the production model, and how it should be consumed.

6.7.7 Write Efficient Code, Compile, and Parallelize

By writing fast and efficient code, you can speed up the training by an order of magnitude, as compared to an inefficient quick-and-dirty script you implemented during experimentation, just "to make it work." Modern datasets are large, so you might wait for hours, even days, for data preprocessing. Training also can take days, or sometimes weeks.

Always write the code with efficiency in mind, even if it seems to be a function, a method, or a script that you will not run frequently. Some code that was supposed to run once might be called in a loop millions of times.

Avoid using loops. For example, if you need to compute a **dot product** of two vectors, or multiply a matrix by a vector, use fast and efficient dot-product or matrix-multiplication methods in scientific libraries and modules. Examples of such efficient implementations are Python's **NumPy** and **SciPy** libraries. Talented and skilled software engineers and scientists created these libraries and modules. They rely on low-level programming languages such as C, as well as hardware acceleration, and work blazingly fast.

Where possible, compile the code before executing it. Such libraries as **PyPy** and **Numba** for Python, or **pqR** for R, would compile the code into the OS (operating system) native binary code, which can significantly increase the speed of data processing and model training.

Another important aspect is parallelization. If you work with modern libraries and modules, you can find learning algorithms that exploit multicore CPUs. Some allow GPUs to speed up the training of neural networks and many other models. Training of some models, such as SVM, cannot be effectively parallelized. In such cases, you can still exploit a multicore CPU by running multiple experiments in parallel. Run one experiment for each combination of hyperparameter values, geographical region, or user segment. Furthermore, compute each cross-validation fold in parallel with other folds.

Where possible, use a solid-state drive (SSD) to store the data. Use distributed computing; some implementations of learning algorithms are designed to run in distributed computing environments, such as Spark. Try to put all the needed data into the RAM of your laptop or server. It's not uncommon today for data analysts to work on a server with 512 gigabytes or even one or more terabytes of RAM.

[8] "Hidden Technical Debt in Machine Learning Systems" by Sculley et al. (2015).

By reducing to a minimum the time needed to train a model, you can spend more time tweaking your model, testing data pre-processing ideas, feature engineering, neural network architectures, and other creative activities. The greatest benefit for the machine learning project lies in the human touch and intuition. The more you, as a human, can work instead of waiting, the higher the chances that your machine learning project will be a success.

Reduce **glue code** to a minimum. This how Google engineers put it. Machine learning researchers tend to develop general purpose solutions as self-contained packages. A wide variety of these are available as open-source packages or from in-house code, proprietary packages, and cloud-based platforms. Using generic packages often results in a glue-code system design pattern, in which a massive amount of supporting code is written to get data into and out of general-purpose packages.

Glue code is costly in the long term. It tends to freeze a system to the peculiarities of a specific package. Testing alternatives may become prohibitively expensive. Using a generic package this way inhibits improvements. It becomes harder to take advantage of domain-specific properties, or to tweak the objective function, and to achieve a domain-specific goal. A mature system might become (at most) 5% machine learning code and (at least) 95% glue code. It may be less costly to create a clean native solution, rather than re-use a generic package.

An important strategy for combating glue code is to wrap black-box machine learning packages into common APIs used by the entire organization. Infrastructure becomes more reusable and it reduces the cost of changing packages.

It is recommended to learn to switch between at least two programming languages: one for fast prototyping (like Python) and one for fast implementation (like C++). Modern languages like Go, Kotlin, and Julia may work well for both cases, but at the time of the writing of this book, these two languages have not developed an ecosystem of machine learning projects, as compared to more established counterparts.

6.7.8 Test on Both Newer and Older Data

If you used a data dump from some time ago to create training, validation, and test sets, observe how your model behaves with data collected before and after this period. If it's radically worse, there's a problem.

Data leakage and **distribution shift** could be among the most likely reasons. Recall that data leakage is when information unavailable in the future or in the past was used to engineer a feature. Distribution shift is when properties of the data change over time.

6.7.9 More Data Beats Cleverer Algorithm

When confronted to insufficient model performance, to improve the performance of the model, analysts are often tempted into crafting a more sophisticated learning algorithm or a pipeline.

In practice, however, better results often come from getting more data, specifically, more labeled examples. If designed well, the data labeling process can allow a labeler to produce several thousand training examples daily. It can also be less expensive, compared to the expertise needed to invent a more advanced machine learning algorithm.

6.7.10 New Data Beats Cleverer Features

If, despite adding more training examples and designing clever features, the performance of your model plateaus, think about different information sources.

For example, if you want to predict whether user U will like a news article, try to add historical data about the user U as features. Or cluster all the users, and use the information on the k-nearest users to user U as new features. This is a simpler approach compared to programming very complex features, or combining existing features in a complex way.

6.7.11 Embrace Tiny Progress

Many tiny improvements to your model may give the expected result faster than looking for one revolutionary idea.

Furthermore, by trying different ideas, the analyst gets to know the data better, which might indeed help in finding that revolutionary idea.

6.7.12 Facilitate Reproducibility

When delivering the model, make sure it's accompanied by all relevant information for **reproducibility**. Besides the description of the dataset and features, such as documentation and metadata considered in Sections **??** and **??**, each model should contain the documentation with the following details:

- a specification of all hyperparameters, including the ranges considered, and the default values used,
- the method used to select the best hyperparameter configuration,
- the definition of the specific measure or statistics used to evaluate the candidate models, and the value of it for the best model,
- a description of the computing infrastructure used, and
- the average runtime for each trained model, and estimated cost of the training.

6.8 Summary

The deep model training strategy has more moving parts, as compared to training shallow models. At the same time, it's more principled and amenable to automation.

Instead of training your model from scratch, it can be useful to start with a pre-trained model. Organizations with access to big data have trained and open-sourced very deep neural networks with architectures optimized for image or natural language processing tasks.

A pre-trained model can be used in two ways: 1) its learned parameters can be used to initialize your own model, or 2) it can be used as a feature extractor for your model.

Using a pre-trained model to build your own is called transfer learning. The fact that deep models allow for transfer learning is one of the most important properties of deep learning.

Minibatch stochastic gradient descent and its variants are the most frequently used cost function optimization algorithms for deep models.

The backpropagation algorithm computes the partial derivatives of each deep model parameter, using the chain rule for derivatives of complex functions. At each epoch, gradient descent updates all parameters using partial derivatives. The learning rate controls the significance of an update. The process continues until convergence, the state where parameters' values don't change much after each epoch. Then the algorithm stops.

There are several popular upgrades to minibatch stochastic gradient descent, such as Momentum, RMSProp, and Adam. These algorithms update the learning rate automatically, based on the performance of the learning process. You do not need to choose the initial value of the learning rate, the decay schedule and rate, or the values of other related hyperparameters. These algorithms have demonstrated good performance in practice, and practitioners often use them instead of trying to manually tune the learning rate.

In addition to L1 and L2 regularization, neural networks benefit from neural network-specific regularizers: dropout, early stopping, and batch-normalization. Dropout is a simple but very effective regularization method. Using batch-normalization is a best practice.

Ensemble learning is training an ensemble model, which is a combination of several base models, each individually performing worse than the ensemble model. There are ensemble learning algorithms, such as random forest and gradient boosting, that build an ensemble of several hundred to thousands of weak models, and obtain a strong model that has a significantly better performance than the performance of each weak model.

Strong models can be combined into an ensemble model by averaging their outputs (for regression) or by taking a majority vote (for classification). Model stacking, being the most effective of the ensembling methods, consists of training a meta-model that takes the output of base models as input.

In addition to using over- and undersampling, imbalanced learning problems can be solved by applying class weighting and ensemble of resampled datasets. If you train your model using stochastic gradient descent, the class imbalance can be tackled in two additional ways: 1) by setting different learning rates for different classes, and 2) by making several consecutive updates of the model parameters each time you encounter an example of a minority class.

For imbalanced learning problems, the performance of the model is measured using adapted performance metrics such as per-class accuracy and Cohen's kappa statistic.

Troubleshooting a machine learning pipeline can be hard. Poor performance can be caused by a bug in your code, training data errors, learning algorithm issues, or pipeline design. In addition, learning can be sensitive to small changes in hyperparameters and dataset makeup.

Errors made by a machine learning model can be uniform and appear in all use cases with the same rate, or focused and appear in just certain types of use cases.

Focused errors are those that merit special attention, because by fixing an error pattern, you fix it once for many examples.

The performance of the model can be iteratively improved using the following simple process:

1. Train the model using the best values of hyperparameters identified so far.
2. Test the model by applying it to a small subset of the validation set ($100-300$ examples).
3. Find the most frequent error patterns on that small validation set. Remove those examples from the validation set, because your model will now overfit to them.
4. Generate new features, or add more training data to fix the observed error patterns.
5. Repeat until no frequent error patterns are observed (most errors look dissimilar).

In complex machine learning systems, the error analysis is done by parts. We first substitute the predictions of one model for the perfect labels (such as human-provided labels), and see how the performance of the entire system improves. If it improves significantly, then more effort must be put in improving that specific model.