Examples II

Eugene Morozov

(Eugene@HiEugene.com)

Contents

Part II
Natural Language Processing / Natural Language Understanding

6 NLP/NLU

6.1 Entropy

Let $p(x)$ be the probability mass function of a random variable X, over a discrete set of symbols (or alphabet) $\mathcal{X}$:

$$p(x) = P(X = x),\ x \in \mathcal{X}$$

The entropy (or self-information) is the average uncertainty of a single random variable:

$$H(p) = H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) = \sum_{x \in \mathcal{X}} p(x) \log_2 \frac{1}{p(x)} \tag{6.1}$$

Entropy measures the amount of information in a random variable. It is normally measured in bits (hence the log to the base 2), but using any other base yields only a linear scaling of results. Also, for this definition to make sense, we define $0 \log 0 = 0$. It makes more sense to interpret (6.1) as

$$H(X) = \mathbb{E}\left[\log_2 \frac{1}{p(x)}\right]$$

One can also think of entropy in terms of the *Twenty Questions* game. For example, for a dice if you ask yes/no questions like "Is it over 3?" or "Is it an even number?" then on average you will need to ask 2.6 questions to identify each number with total certainty (assuming that you ask good questions!). In other words, entropy can be interpreted as a measure of the size of the "search

space" consisting of the possible values of a random variable and its associated probabilities.

The joint entropy of a pair of discrete random variables $X, Y \sim p(x, y)$ is the amount of information needed on average to specify both their values. It is defined as:

$$H(X, Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log_2 p(x, y)$$

The **conditional entropy** of a discrete random variable Y given another X, for $X, Y \sim p(x, y)$, expresses how much extra information you still need to supply on average to communicate Y given that the other party knows X:

$$\begin{aligned} H(Y|X) &= \sum_{x \in \mathcal{X}} p(x) H(Y|X = x) \\ &= \sum_{x \in \mathcal{X}} p(x) \left[-\sum_{y \in \mathcal{Y}} p(y|x) \log_2 p(y|x) \right] \\ &= -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log_2 p(y|x) \end{aligned} \tag{6.2}$$

By the chain rule for entropy,

$$H(X, Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

Therefore,

$$H(X) - H(X|Y) = H(Y) - H(Y|X)$$

This difference is called the **mutual information** between X and Y. It is the reduction in uncertainty of one random variable due to knowing about another, or in other words, the amount of information one random variable contains about another.

$$\begin{aligned} I(X; Y) &= H(X) - H(X|Y) \\ &= H(X) + H(Y) - H(X, Y) \\ &= \sum_{x,y} p(x, y) \log_2 \frac{p(x, y)}{p(x)p(y)} \\ &= \mathbb{E}_{p(x,y)} \left[\log_2 \frac{p(x, y)}{p(x)p(y)} \right] \\ &= KL\left(p(x, y) \| p(x)p(y)\right) \end{aligned} \tag{6.3}$$

Consider Simplified Polynesian language. This language has 6 letters. The simplest code is to use 3 bits for each letter of the language. This is equivalent to assuming that a good model of the language (where our 'model' is simply a probability distribution) is a uniform model. However, we noticed that not all the letters occurred equally often, and, noting these frequencies, produced a

zeroth order model of the language. This had a lower entropy of 2.5 bits per letter (and we showed how this observation could be used to produce a more efficient code for transmitting the language). Thereafter, we noticed the syllable structure of the language, and developed an even better model that incorporated that syllable structure into it. The resulting model had an even lower entropy of 1.22 bits per letter. The essential point here is that if a model captures more of the structure of a language, then the entropy of the model should be lower. In other words, we can use entropy as a measure of the quality of our models.

Statistical NLP problems as decoding problems:

Application	Input	Output	$p(i)$	$p(o\|i)$
Machine Translation	L_1 word sequences	L_2 word sequences	$p(L_1)$ in a language model	translation model
Optical Character Recognition (OCR)	actual text	text with mistakes	prob of language text	model of OCR errors
Part Of Speech (POS) tagging	POS tag sequences	English words	prob of POS sequences	$p(w\|t)$
Speech recognition	word sequences	speech signal	prob of word sequences	acoustic model

n-gram models represent k^{th} order Markov chain approximation:

$$\begin{aligned} &P(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_1 = x_1) \\ \approx\ &P(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_{n-k} = x_{n-k}) \end{aligned}$$

For the probability of occurrence of particular words w_1 and w_2 we can define

$$\begin{aligned} I(w_1; w_2) &= \log_2 \frac{P(w_1 w_2)}{P(w_1)P(w_2)} \\ &= \log_2 \frac{P(w_1|w_2)}{P(w_1)} \\ &= \log_2 \frac{P(w_2|w_1)}{P(w_2)} \end{aligned} \tag{6.4}$$

where $P(w_1) = \mathit{count\ of\ word}\, w_1 \,/\, \mathit{total\ number\ of\ words}$, and for $P(w_1 w_2)$ we use the collocation frequency, i.e. bigram. In information theory, mutual information is more often defined as holding between random variables (cf (6.3)), not values of random variables as we have defined it here. The 2nd equation in (6.4) gives us the amount of information provided by the occurrence of the event represented by $[w_2]$ about the occurrence of the event represented by $[w_1]$. For example, the mutual information measure tells us that the amount of information we have about the occurrence of "Prime" at position i in the corpus

increases by certain number of bits if we are told that "Minister" occurs at position $i+1$. We could also say that our uncertainty is reduced by that number of bits.

Information radius

$$IRad = KL\left(p\|\frac{p+q}{2}\right) + KL\left(q\|\frac{p+q}{2}\right)$$

(or total divergence to the average) is symmetric ($IRad(p,q) = IRad(q,p)$) and there is no problem with infinite values since $\frac{p_i+q_i}{2} \neq 0$ if either $p_i \neq 0$ or $q_i \neq 0$. The intuitive interpretation of $IRad$ is that it answers the question: How much information is lost if we describe the two words (or random variables in the general case) that correspond to p and q with their average distribution? $IRad$ ranges from 0 for identical distributions to $2\log 2$ for maximally different distributions. As usual we assume $0\log 0 = 0$.

Word (or term) occurrence frequency can be used to represent a document as a vector in n-dimensional word (phrase) space. In this space "close" often means small angle between vectors (without magnitude). To search for collocations the tuple: (word occurrence frequency, position), where position is simply an offset from beginning, can be used.

Cosine measure or normalized correlation coefficient

$$cos(\overrightarrow{q}, \overrightarrow{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2}\sqrt{\sum_{i=1}^{n} d_i^2}}$$

where $\overrightarrow{q}$ is the query vector, e.g. "car insurance", $\overrightarrow{d}$ is a document vector.

An interesting property of the cosine is that, if applied to normalized vectors, it will give the same ranking of similarities as Euclidean distance does.

$$\begin{aligned}
|\overrightarrow{x} - \overrightarrow{y}|^2 &= \sum_{i=1}^{n}(x_i - y_i)^2 \\
&= \sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_i y_i + \sum_{i=1}^{n} y_i^2 \\
&= 1 - 2\sum_{i=1}^{n} x_i y_i + 1 \\
&= 2(1 - \sum_{i=1}^{n} x_i y_i)
\end{aligned}$$

So for a particular query $\overrightarrow{q}$ and any 2 documents $\overrightarrow{d}_1$ and $\overrightarrow{d}_2$ we have:

$$cos(\overrightarrow{q}, \overrightarrow{d}_1) > cos(\overrightarrow{q}, \overrightarrow{d}_2) \Leftrightarrow |\overrightarrow{q} - \overrightarrow{d}_1| < |\overrightarrow{q} - \overrightarrow{d}_2|$$

If the vectors are normalized (i.e. $\sqrt{\sum_{i=1}^{n} d_i^2} = 1$), we can compute the cosine as a simple dot product. Normalization is generally seen as a good thing

– otherwise longer vectors (corresponding to longer documents) would have an unfair advantage and get ranked higher than shorter ones.

Term weighting:

quantity	symbol	definition
term frequency	$tf_{i,j}$	number of occurrences of w_i in d_j
document frequency	df_i	number of documents in the collection that w_i occurs in
collection frequency	cf_i	total number of occurrences of w_i in the collection

Term frequency is usually dampened by a function like $f(tf) = \sqrt{tf}$ or $f(tf) = 1 + \log(tf)$, $tf > 0$ because more occurrences of a word indicate higher importance, but not as much importance as the undampened count would suggest.

One way to combine a word's term frequency $tf_{i,j}$ and document frequency df_i into a single weight is as follows:

$$weight(i,j) = \begin{cases} (1 + \log tf_{i,j}) \log \frac{N}{df_i} & if\ tf_{i,j} \geq 1 \\ 0 & if\ tf_{i,j} = 0 \end{cases}$$

where N is the total number of documents. This is so-called inverse document frequency (idf) weighting. Thus, it produces *term frequency - inverse document frequency* (tf.idf). There are several weighting configurations.

It can be shown that mutual information between random variables $\mathcal{T}$ and $\mathcal{D}$ corresponding to respectively drawing a document or a term

$$I(\mathcal{T};\mathcal{D}) = \frac{1}{|D|} \sum_{t,d} tf(t,d) \cdot idf(t)$$

This expression shows that summing the tf.idf of all possible terms and documents recovers the mutual information between documents and term taking into account all the specificities of their joint distribution. Each tf.idf hence carries the "bit of information" attached to a term w document pair.

The probability that a word w_i occurs a particular number of times k in a document can be modeled (although sometimes inaccurately) by the Poisson distribution:

$$p(k;\lambda_i) = e^{-\lambda_i} \frac{\lambda_i^k}{k!}$$

where $\lambda_i > 0$ is the average number of occurrences of w_i per document, that is, $\lambda_i = cf_i/N$ where N is the total number of documents in the collection.

A simpler distribution that fits empirical word distributions about as well as the negative binomial is Katz's K mixture:

$$P_i(k) = (1-\alpha)\delta_{k,0} + \frac{\alpha}{\beta+1}\left(\frac{\beta}{\beta+1}\right)^k$$

where $\delta_{k,0} = 1$ iff $k = 0$ and $\delta_{k,0} = 0$ otherwise and α and β are parameters that can be fit using the observed mean λ and the observed inverse document frequency idf as follows.

$$\lambda = \frac{cf}{N}$$
$$idf = \log_2 \frac{N}{df}$$
$$\beta = \lambda \times 2^{idf} - 1 = \frac{cf - df}{df}$$
$$\alpha = \frac{\lambda}{\beta}$$

The parameter β is the number of "extra terms" per document in which the term occurs (compared to the case where a term has only one occurrence per document). The decay factor $\frac{\beta}{\beta+1} = \frac{cf-df}{cf}$ (extra terms per term occurrence) determines the ratio $\frac{P_i(k)}{P_i(k-1)}$. For example, if there are $1/10$ as many extra terms as term occurrences, then there will be ten times as many documents with 1 occurrence as with 2 occurrences and ten times as many with 2 occurrences as with 3 occurrences. If there are no extra terms ($cf = df \Rightarrow \frac{\beta}{\beta+1} = 0$), then we predict that there are no documents with more than 1 occurrence.

The parameter α captures the absolute frequency of the term. Two terms with the same β have identical ratios of collection frequency to document frequency, but different values for α if their collection frequencies are different.

6.2 Latent Semantic Indexing

LSI is a technique that projects queries and documents into a space with "latent" semantic dimensions. In the latent semantic space, a query and a document can have high cosine similarity even if they do not share any terms – as long as their terms are semantically similar according to the co-occurrence analysis. We can look at LSI as a similarity metric that is an alternative to word overlap measures like tf.idf.

Latent Semantic Indexing is closely related to Principal Component Analysis (PCA), another technique for dimensionality reduction. Take a term-by-document matrix (where rows are words and columns are documents; values are weighted occurrences) and perform an SVD. SVD (and hence LSI) is a least-squares method. The projection into the latent semantic space is chosen such that the representations in the original space are changed as little as possible when measured by the sum of the squares of the differences.

One objection to SVD is that, along with all other least-squares methods, it is really designed for normally-distributed data. But often other distributions like Poisson or negative binomial are more appropriate for term counts. One problematic feature of SVD is that, since the reconstruction $\hat{A}$ of the term-by-document matrix A is based on a normal distribution, it can have negative entries, clearly an inappropriate approximation for counts. A dimensionality reduction based on Poisson would not predict such impossible negative counts.

△ **Wikipedia**.

Out of 10,460,720 English Wikipedia articles I've selected 288 articles in 2 areas: statistics and topology. They contain 14,427 terms. After performing SVD $K = 20$ dimensions were retained, i.e. the term-by-documents matrix A is represented as

$$A = TSD^T \tag{6.5}$$

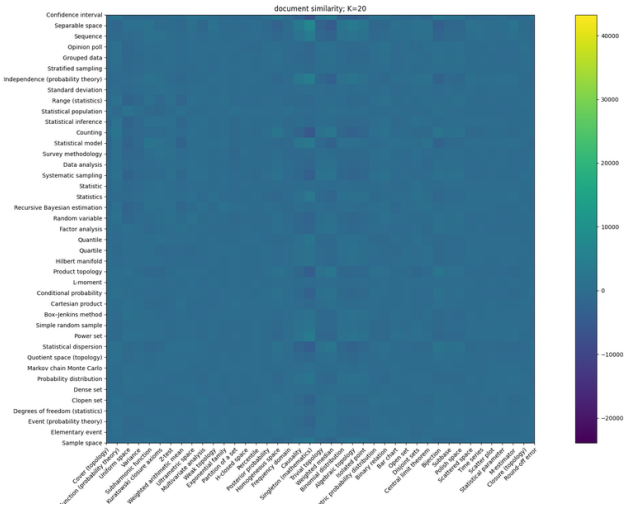
where T is the term matrix and D is the document matrix.

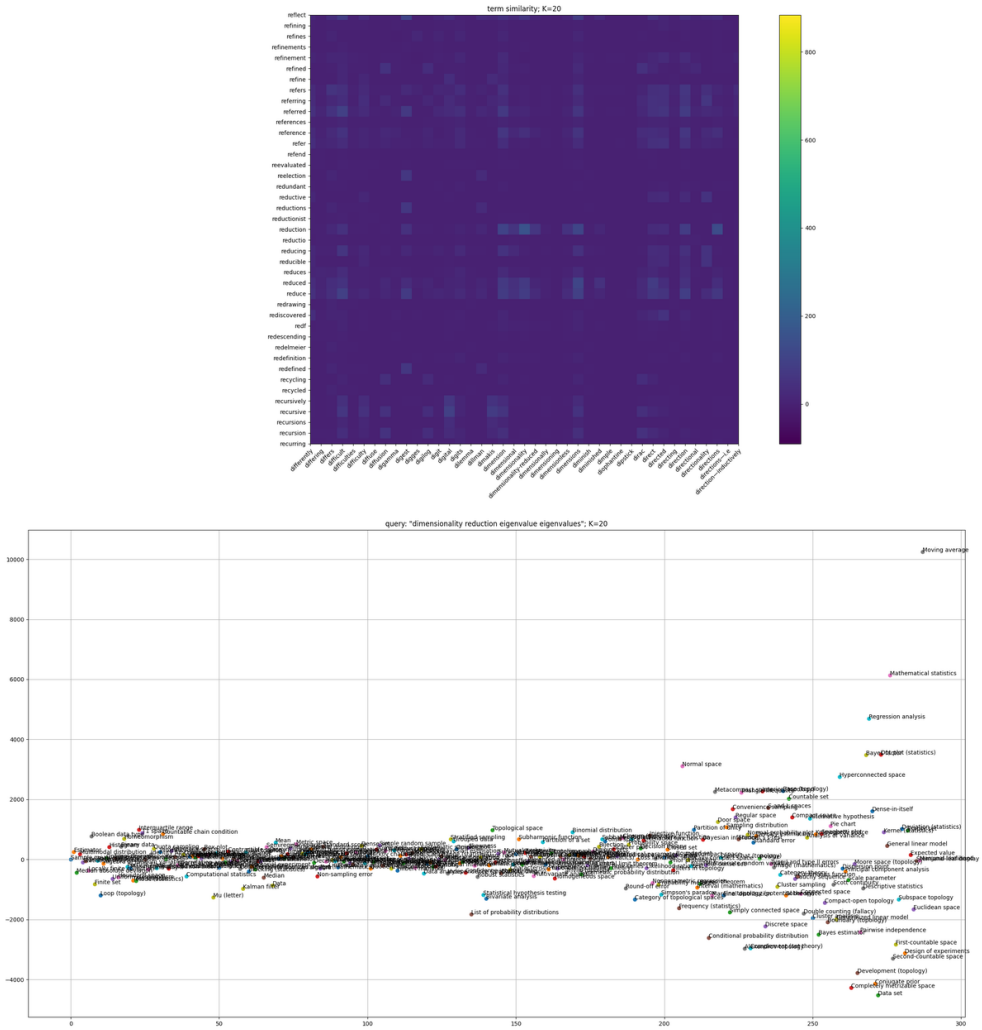
A query can be thought as a mini document. The search is based on (6.5) being equivalent to

$$T^T A = SD^T$$

So we just multiply the query or document vector with the transpose of the term matrix T (after it has been truncated to the desired dimensionality). For example, for a query vector $\overrightarrow{q}$ and a reduction to dimensionality k, the query representation in the reduced space is $T^T_{m \times k} \overrightarrow{q}$. In other words, we project the query to the reduced dimensionality space and find the closest (by dot product) documents.

$(DS)(DS)^T$ gives us the correlation or document similarity; the same is for term similarity: $(TS)(TS)^T$.





6.3 Classification Models

Contingency table for evaluating a binary classifier:

	YES is correct	NO is correct
YES was assigned	a	b
NO was assigned	c	d

then

accuracy	$\frac{a+d}{a+b+c+d}$
precision	$\frac{a}{a+b}$
recall	$\frac{a}{a+c}$
fallout	$\frac{b}{b+d}$

6.3.1 Decision Trees

For text categorization, the information retrieval vector space model is frequently used as the data representation. That is, each document is represented as a vector of (possibly weighted) word counts.

1. Pick $K = 20$ words whose χ^2-test score is the highest for a category.
2. Each document is then represented as a vector of K integers, $\overrightarrow{x}_j = (s_{1j}, \ldots, s_{Kj})$, where s_{ij} is computed as the following quantity:

$$s_{ij} = 10 \frac{1 + \log tf_{ij}}{1 + \log l_j}$$

   where tf_{ij} is the number of occurrences of term i in document j and l_j is the length of document j. The score s_{ij} is set to 0 for no occurrences of the term.

3. Use *grow* and *prune* technique to train the model. The *splitting criterion* which we can use is to split the objects at a node into two piles in the way that gives us maximum information gain. **Information gain** is an information-theoretic measure defined as the difference of the entropy of the mother node and the weighted sum of the entropies of the child nodes:

$$G(a, y) = H(t) - H(t|a) = H(t) - (p_L H(t_L) + p_R H(t_R))$$

   where a is the attribute we split on, y is the value of a we split on, t is the distribution of the node that we split, p_L and p_R are the proportion of elements that are passed on to the left and right nodes, and t_L and t_R are the distributions of the left and right nodes. Information gain is intuitively appealing because it can be interpreted as measuring the reduction of uncertainty.

6.3.2 Maximum Entropy Modeling

This term may initially seem perverse, since we have spent most of the book trying to minimize the (cross) entropy of models, but the idea is that we do not want to go beyond the data. If we chose a model with less entropy, we would add "information" constraints to the model that are not justified by the empirical evidence available to us. Choosing the maximum entropy model is motivated by the desire to preserve as much uncertainty as possible. Somewhat simplified scheme:

1. Select features. We first compute the expectation of each feature based on the training set. Each feature then defines the constraint that this empirical expectation be the same as the expectation the feature has in our final maximum entropy model.

2. Select models (family of probability distributions). For example, log-linear models

$$p(\overrightarrow{x}, c) = \frac{1}{Z} \prod_{i=1}^{K} \alpha_i^{f_i(\overrightarrow{x}, c)}$$

where K is the number of features, α_i is the weight for feature f_i, and Z is a normalizing constant, used to ensure that a probability distribution results. The features f_i may be binary functions that can be used to characterize any property of a pair $(\overrightarrow{x}, c)$, where $\overrightarrow{x}$ is a vector representing an input element (e.g. the same 20-dimensional vector of words as in the previous subsection), and c is the class label (1 if the article is in the "earnings" category, 0 otherwise). For text categorization, we can define features as follows:

$$f_i(\overrightarrow{x}_j, c) = \begin{cases} 1 & if\, s_{ij} > 0\, and\, c = 1 \\ 0 & otherwise \end{cases}$$

3. Of all probability distributions that obey these constraints, we attempt to find the maximum entropy distribution p^*, the one with the highest entropy. One can show that there is a unique such maximum entropy distribution and there exists an algorithm, generalized iterative scaling, which is guaranteed to converge to it.

   (a) p^* obeys the following set of constrains:

$$E_{p^*} f_i = E_{\tilde{p}} f_i \tag{6.6}$$

   In other words, the expected value of f_i for p^* is the same as the expected value for the empirical distribution (in other words, the training set).

   (b) The algorithm requires that the sum of the features for each possible $(\overrightarrow{x}, c)$ be equal to a constant C:

$$\forall\ \overrightarrow{x}, c\ \sum_i f_i\ (\overrightarrow{x}, c) = C$$

   In order to fulfill this requirement, we define C as the greatest possible feature sum:

$$C := \max_{\overrightarrow{x}, c} \sum_{i=1}^{K} f_i(\overrightarrow{x}, c)$$

and add a feature f_{K+1} that is defined as follows:

$$f_{K+1}(\overrightarrow{x}, c) = C - \sum_{i=1}^{K} f_i(\overrightarrow{x}, c)$$

Note that this feature is not binary, in contrast to the others.

(c) $E_p f_i$ is defined as follows:

$$E_p f_i = \sum_{\overrightarrow{x}, c} p(\overrightarrow{x}, c) f_i(\overrightarrow{x}, c)$$

where the sum is over the event space, that is, all possible vectors $\overrightarrow{x}$ class labels c.

(d) The empirical expectation is easy to compute:

$$E_{\bar{p}} f_i = \sum_{\overrightarrow{x}, c} \bar{p}(\overrightarrow{x}, c) f_i(\overrightarrow{x}, c) = \frac{1}{N} \sum_{j=1}^{K} f_i(\overrightarrow{x}_j, c)$$

where N is the number of elements in the training set and we use the fact that the empirical probability for a pair that doesn't occur in the training set is 0.

(e) In general, the maximum entropy distribution $E_p f_i$ cannot be computed efficiently since it would involve summing over all possible combinations of $\overrightarrow{x}$ and c, a potentially infinite set. Instead, we use the following approximation:

$$E_p f_i = \frac{1}{N} \sum_{j=1}^{N} \sum_{c} p(c|\overrightarrow{x}_j) f_i(\overrightarrow{x}_j, c)$$

where c ranges over all possible classes, in our case $c \in 0, 1$.

(f) Now we have all the pieces to state the generalized **iterative scaling algorithm**:

i. Initialize $\alpha_i^{(1)}$. Any initialization will do, but usually we choose $\alpha_i^{(1)} = 1, \forall 1 \leq j \leq K+1$. Compute $E_{\bar{p}} f_i$ as shown above. Set $n = 1$.

ii. Compute $p^{(n)}(\overrightarrow{x}, c)$ for the distribution $p^{(n)}$ given by the $\{\alpha_i^{(n)}\}$ for each element $(\overrightarrow{x}, c)$ in the training set:

$$p^{(n)}(\overrightarrow{x}, c) = \frac{1}{Z} \prod_{i=1}^{K+1} \left(\alpha_i^{(n)}\right)^{f_i(x,c)}$$

iii. Compute $E_{p^{(n)}} f_i$ for all $1 \leq i \leq K+1$.

iv. Update the parameters α_i:

$$\alpha_i^{(n+1)} = \alpha_i^{(n)} \left(\frac{E_{\bar{p}} f_i}{E_{p^{(n)}} f_i}\right)^{\frac{1}{c}}$$

v. If the parameters of the procedure have converged, stop, otherwise increment n and go to 2.

vi. Note: in an actual implementation, it is more convenient to do the computations using logarithms. One can show that this procedure converges to a distribution p^* that obeys the constraints (6.6), and that of all such distributions it is the one that maximizes the entropy $H(p)$ and the likelihood of the data.

4. We can use $P(\text{“}earnings\text{”}|\overrightarrow{x}) > P(\neg\text{“}earnings\text{”}|\overrightarrow{x})$ as our decision rule.

6.3.3 Perceptrons

As before, text documents are represented as term vectors. Our goal is to learn a weight vector $\overrightarrow{w}$ and a threshold θ, such that comparing the dot product of the weight vector and the term vector against the threshold provides the categorization decision. We decide “yes” (the article is in the “earnings” category) if the inner product of weight vector and document vector is greater than the threshold and “no” otherwise:

$$decide\text{”}yes\text{”}\ iff \quad \overrightarrow{w} \cdot \overrightarrow{x} = \sum_{i=1}^{K} w_i x_{ij} > 0$$

where K is the number of features ($K = 20$ for our example as before) and x_{ij} is component i of vector $\overrightarrow{x}_j$.

The basic idea of the perceptron learning algorithm is simple. If the weight vector makes a mistake, we move it (and θ) in the direction of greatest change for our optimality criterion $\sum_{i=1}^{K} w_i x_{ij} - \theta$. To see that the changes to $\overrightarrow{w}$ and θ in figure 16.8 are made in the direction of greatest change, we first define an optimality criterion φ that incorporates θ into the weight vector:

$$\varphi(\overrightarrow{w}') = \varphi\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_K \\ \theta \end{pmatrix} = \overrightarrow{w}' \cdot \overrightarrow{x}' = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_K \\ \theta \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \\ -1 \end{pmatrix}$$

The gradient of φ (which is the direction of greatest change) is the vector $\overrightarrow{x}'$:

$$\nabla\varphi(\overrightarrow{w}') = \overrightarrow{x}'$$

Of all vectors of a given length that we can add to $\overrightarrow{w}'$, $\overrightarrow{x}'$ is the one that will change the most.

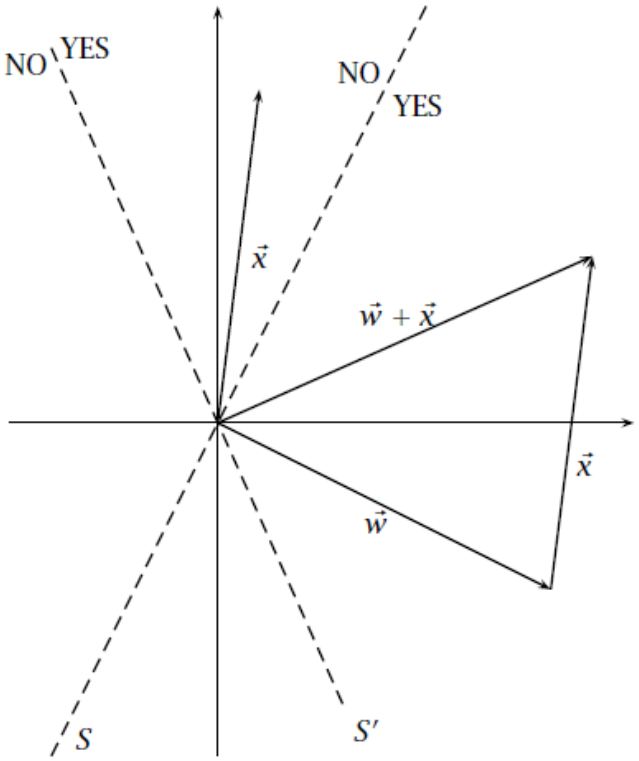


Figure 16.8 One error-correcting step of the perceptron learning algorithm. Data vector $\vec{x}$ is misclassified by the current weight vector $\vec{w}$ since it lies on the "no"-side of the decision boundary S. The correction step adds $\vec{x}$ to $\vec{w}$ and (in this case) corrects the decision since $\vec{x}$ now lies on the "yes"-side of S', the decision boundary of the new weight vector $\vec{w} + \vec{x}$.

The figure also illustrates the class of models that can be learned by perceptrons: linear separators.

Algorithm:

```
// Categorization Decision
fun decision(x⃗,w⃗,θ) =
  if w⃗ · x⃗ > 0 then
    return "yes"
  else
    return "no"
// Initialization
w⃗ = 0
θ = 0
// Perceptron Learning Algorithm
while not converged yet do
  for all elements x⃗_j in the training set do
    d = decision(x⃗_j,w⃗,θ)
    if class(x⃗_j) = d then
      continue
    else if class(x⃗_j) = "yes" and d = "no" then
      θ = θ − 1
      w⃗ = w⃗ + x⃗_j
    else if class(x⃗_j) = "no" and d = "yes" then
      θ = θ + 1
      w⃗ = w⃗ − x⃗_j
    fi
  end
end
```

6.3.4 k Nearest Neighbor Classification

The cosine similarity metric is often used.

6.4 Conditional Random Field

Assuming we have a sequence of input words $X = x_1 \dots x_n$ and want to compute a sequence of output tags $Y = y_1 \dots y_n$. In an HMM to compute the best tag sequence that maximizes $P(Y|X)$ we rely on Bayes' rule and the likelihood $P(X|Y)$:

$$\begin{aligned}\hat{Y} &= \arg\max_Y p(Y|X) \\ &= \arg\max_Y p(X|Y)p(Y) \\ &= \arg\max_Y \prod_i p(x_i|y_i) \prod_i p(y_i|y_{i-1})\end{aligned}$$

In a CRF, by contrast, we compute the posterior $P(Y|X)$ directly, training the CRF to discriminate among the possible tag sequences:

$$\hat{Y} = \arg\max_{Y \in \mathcal{Y}} P(Y|X)$$

However, the CRF does not compute a probability for each tag at each time step. Instead, at each time step the CRF computes log-linear functions over a set of relevant features, and these local features are aggregated and normalized to produce a global probability for the whole sequence.

A CRF is a log-linear model that assigns a probability to an entire output (tag) sequence Y, out of all possible sequences $\mathcal{Y}$, given the entire input (word) sequence X. We can think of a CRF as like a giant version of what multinomial logistic regression does for a single token. Recall that the feature function f in regular multinomial logistic regression maps a tuple of a token x and a label y into a feature vector. In a CRF, the function F maps an entire input sequence X and an entire output sequence Y to a feature vector. Let's assume we have K features, with a weight w_k for each feature F_k:

$$p(Y|X) = \frac{\exp\left(\sum_{k=1}^{K} w_k F_k(X,Y)\right)}{\sum_{Y' \in \mathcal{Y}} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y')\right)}$$

It's common to also describe the same equation by pulling out the denominator into a function $Z(X)$:

$$p(Y|X) = \frac{1}{Z(X)} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y)\right)$$

$$Z(X) = \sum_{Y' \in \mathcal{Y}} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y')\right)$$

We'll call these K functions $F_k(X,Y)$ global features, since each one is a property of the entire input sequence X and output sequence Y. We compute them by decomposing into a sum of local features for each position i in Y:

$$F_k(X,Y) = \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i)$$

Each of these local features f_k in a linear-chain CRF is allowed to make use of the current output token y_i, the previous output token y_{i-1}, the entire input string X (or any subpart of it), and the current position i. This constraint to only depend on the current and previous output tokens y_i and y_{i-1} are what characterizes a linear chain CRF. This limitation makes it possible to use versions of the efficient Viterbi and Forward-Backwards algorithms from the HMM. A general CRF, by contrast, allows a feature to make use of any output token, and are thus necessary for tasks in which the decision depend on distant output tokens, like y_{i-4}. General CRFs require more complex inference, and are less commonly used for language processing.

Examples for some features include:

$$\mathbb{1}\{x_i = the,\, y_i = DET\}$$
$$\mathbb{1}\{y_i = PROPN,\, x_{i+1} = Street,\, y_{i-1} = NUM\}$$

To decode (finding the most likely tag sequence) we use he Viterbi algorithm. This involves filling an $N \times T$ array with the appropriate values, maintaining backpointers as we proceed. As with HMM Viterbi, when the table is filled, we simply follow pointers back from the maximum value in the final column to retrieve the desired set of labels. A cell value of time t for state j

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) \sum_{k=1}^{K} w_k f_k(y_{t-1}, y_t, X, t) \quad 1 \leq j \leq N, 1 < t \leq T$$

Learning in CRFs relies on the same supervised learning algorithms as for logistic regression. Given a sequence of observations, feature functions, and corresponding outputs, we use stochastic gradient descent to train the weights to maximize the log-likelihood of the training corpus. The local nature of linear-chain CRFs means that a CRF version of the forward-backward algorithm can be used to efficiently compute the necessary derivatives. As with logistic regression, L1 or L2 regularization is important.

6.5 Naive Bayes for words disambiguation

Bayes decision rule:

decide s' *if* $P(s'|c) > P(s_k|c)$ *for* $s_k \neq s'$

where $s_1, \ldots, s_K$ senses of an ambiguous word w, $c_1, \ldots, c_I$ contexts of w in a corpus, $v_1, \ldots, v_J$ words used as contextual features for disambiguation.

We usually do not know the value of $P(s_k|c)$, but we can compute it using Bayes' rule:

$$P(s_k|c) = \frac{P(c|s_k)}{P(c)} P(s_k)$$

$P(s_k)$ is the prior probability of sense s_k, the probability that we have an instance of s_k if we do not know anything about the context. $P(s_k)$ is updated with the factor $\frac{P(c|s_k)}{P(c)}$ which incorporates the evidence which we have about the context, and results in the posterior probability $P(s_k|c)$.

If all we want to do is choose the correct class then

$$\begin{aligned} s' &= \arg\max_{s_k} P(s_k|c) \\ &= \arg\max_{s_k} \frac{P(c|s_k)}{P(c)} P(s_k) \\ &= \arg\max_{s_k} P(c|s_k) P(s_k) \\ &= \arg\max_{s_k} [\log P(c|s_k) + \log P(s_k)] \end{aligned}$$

The Naive Bayes assumption is that the attributes used for description are all conditionally independent:

$$P(c|s_k) = P\left(\{v_j|v_j \, in \, c\}|s_k\right) = \prod_{v_j \, in \, c} P(v_j|s_k)$$

In our case, the Naive Bayes assumption has two consequences. The first is that all the structure and linear ordering of words within the context is ignored. This is often referred to as a bag of words model. The other is that the presence of one word in the bag is independent of another. This is clearly not true.

Decision rule for Naive Bayes:

decide $s' \; if \; s' = \arg\max_{s_k} \left[\log P(s_k) + \sum_{v_j \, in \, c} \log P(v_j|s_k)\right]$

$P(v_j|s_k)$ and $P(s_k)$ are computed via Maximum-Likelihood estimation, perhaps with appropriate smoothing, from the labeled training corpus:

$$P(v_j|s_k) = \frac{C(v_j, s_k)}{C(s_k)}$$

$$P(s_k) = \frac{C(s_k)}{C(w)}$$

where $C(v_j, s_k)$ is the number of occurrences of v_j in a context of sense s_k in training corpus, $C(s_k)$ is the number of occurrences of s_k in the training corpus, and $C(w)$ is the total number of occurrences of the ambiguous word w.

6.6 Word2vec

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. The vectors are chosen carefully such that the cosine similarity between the vectors indicates the level of semantic similarity between the words represented by those vectors.

For a sentence of n words $w_1, \ldots, w_n$, contexts of a word w_i comes from a window of size k around the word: $C(w) = w_{i-k}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+k}$, where k is a parameter. [6] used a dynamic window size $k' \leq k$ with k' sampled uniformly for each word in the corpus.

The revolutionary intuition here is that we can just use running text as implicitly supervised training data for such a classifier; a word c that occurs near the target word *apricot* acts as gold "correct answer" to the question "Is word c likely to show up near *apricot*?" This method, often called self-supervision.

The continuous skip-gram model is similar to the continuous bag-of-words model, but instead of predicting the current word based on the context, it tries to maximize classification of a word based on another word in the same sentence. More precisely, we use each current word as an input to a log-linear classifier with continuous projection layer, and predict words within a certain range before and after the current word. In the skip-gram and ivLBL models, the objective is to predict a word's context given the word itself, whereas the objective in the CBOW and vLBL models is to predict a word given its context.

In the skip-gram model we are given a corpus of words w and their contexts c. We consider the conditional probabilities $p(c|w)$, and given a corpus T (Text), the goal is to set the parameters θ of $p(c|w;\theta)$ so as to maximize the corpus probability (with the assumption that all context words are independent):

$$\arg\max_{\theta} \prod_{w \in T} \left[\prod_{c \in C(w)} p(c|w;\theta) \right]$$

in this equation, $C(w)$ is the set of contexts of word w. Alternatively:

$$\arg\max_{\theta} \prod_{(w,c) \in D} p(c|w;\theta) \tag{6.7}$$

here D is the set of all word and context pairs we extract from the text.

So, skip-gram actually stores two embeddings for each word, one for the word as a target, and one for the word considered as context. Thus the parameters we need to learn are two matrices W and C, each containing an embedding for every one of the $|V|$ words in the vocabulary V.

One approach for parameterizing the skip-gram model follows the neural-network language models literature, and models the conditional probability $p(c|w;\theta)$ using soft-max:

$$p(c|w;\theta) = \frac{e^{v_c v_w}}{\sum_{c' \in C} e^{v_{c'} v_w}}$$

where v_c and $v_w \in \mathbb{R}^d$ are vector representations for c and w respectively, and C is the set of all available contexts. The parameters θ are v_{c_i}, v_{w_i} for $w \in V$, $c \in C$, $i \in 1, \ldots, d$ (a total of $|C| \times |V| \times d$ parameters). We would like to set the parameters such that the product (6.7) is maximized.

Taking the log switches from product to sum:

$$\arg\max_{\theta} \prod_{(w,c) \in D} \log p(c|w;\theta) = \sum_{(w,c) \in D} \left(\log e^{v_c v_w} - \log \sum_{c' \in C} e^{v_{c'} v_w} \right) \tag{6.8}$$

An assumption is that maximizing objective (6.8) will result in good embeddings $v_w \; \forall \, w \in V$, in the sense that similar words will have similar vectors. It is not clear to us at this point why this assumption holds. But we try to increase the quantity $v_w \cdot v_c$ for good word-context pairs (after all, cosine is just a normalized dot product), and decrease it for bad ones. Intuitively, this means that words that share many contexts will be similar to each other (note also that contexts sharing many words will also be similar to each other).

While objective (6.8) can be computed, it is computationally expensive to do so, because the term $p(c|w;\theta)$ is very expensive to compute due to the summation $\sum_{c' \in C} e^{v_{c'} v_w}$ over all the contexts $c\prime$ (there can be hundreds of thousands of them). One way of making the computation more tractable is to replace the softmax with a hierarchical softmax.

[6] presents the negative-sampling approach as a more efficient way of deriving word embeddings. While negative-sampling is based on the skip-gram model, it is in fact optimizing a different objective.

Consider a pair (w, c) of word and context. Did this pair come from the training data? Let's denote by $p(D = 1|w, c)$ the probability that (w, c) came from the corpus data. We don't actually care about this binary prediction task; instead we'll take the learned classifier weights as the word embeddings. Correspondingly, $p(D = 0|w, c) = 1 - p(D = 1|w, c)$ will be the probability that (w, c) did not come from the corpus data. As before, assume there are parameters θ controlling the distribution: $p(D = 1|w, c; \theta)$. Our goal is now to find parameters to maximize the probabilities that all of the observations indeed came from the data:

$$\begin{aligned}
&\arg\max_\theta \prod_{(w,c)\in D} p(D = 1|w, c; \theta) \\
&= \arg\max_\theta \log \prod_{(w,c)\in D} p(D = 1|w, c; \theta) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log p(D = 1|w, c; \theta)
\end{aligned}$$

The quantity $p(D = 1|c, w; \theta)$ can be defined using softmax:

$$p(D = 1|c, w; \theta) = \frac{1}{1 + e^{-v_c v_w}}$$

Leading to the objective:

$$\arg\max_\theta \sum_{(w,c)\in D} \log \frac{1}{1 + e^{-v_c v_w}}$$

This objective has a trivial solution if we set θ such that $p(D = 1|w, c; \theta) = 1$ for every pair (w, c). This can be easily achieved by setting θ such that $v_c = v_w$ and $v_c \cdot v_w = K$ for all v_c, v_w, where K is large enough number (practically, we get a probability of 1 as soon as $K \approx 40$).

We need a mechanism that prevents all the vectors from having the same value, by disallowing some (w, c) combinations. One way to do so, is to present the model with some (w, c) pairs for which $p(D = 1|w, c; \theta)$ must be low, i.e. pairs which are not in the data. This is achieved by generating the set D' of random (w, c) pairs, assuming they are all incorrect (the name "negative-sampling" stems from the set D' of randomly sampled negative examples). The optimization objective now becomes:

$$
\begin{aligned}
&\arg\max_\theta \prod_{(w,c)\in D} p(D=1|w,c;\theta) \prod_{(w,c)\in D'} p(D=0|w,c;\theta) \\
&= \arg\max_\theta \prod_{(w,c)\in D} p(D=1|w,c;\theta) \prod_{(w,c)\in D'} (1-p(D=1|w,c;\theta)) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log p(D=1|w,c;\theta) + \sum_{(w,c)\in D'} \log (1-p(D=1|w,c;\theta)) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c v_w}} + \sum_{(w,c)\in D'} \log \left(1-\frac{1}{1+e^{-v_c v_w}}\right) \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c v_w}} + \sum_{(w,c)\in D'} \log \frac{1}{1+e^{v_c v_w}}
\end{aligned}
$$

If we let $\sigma(x) = \frac{1}{1+e^{-x}}$ (to turn the dot product into a probability) we get:

$$
\begin{aligned}
&\arg\max_\theta \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c v_w}} + \sum_{(w,c)\in D'} \log \frac{1}{1+e^{v_c v_w}} \\
&= \arg\max_\theta \sum_{(w,c)\in D} \log \sigma(v_c v_w) + \sum_{(w,c)\in D'} \log \sigma(-v_c v_w)
\end{aligned}
\tag{6.9}
$$

With negative sampling of k, [6] constructed D' is k times larger than D, and for each $(w,c) \in D$ we construct k samples $(w,c_1),\ldots,(w,c_k)$, where each c_j is drawn according to its unigram distribution (frequency in T) raised to the $3/4$ power. In [6] each context is a word (and all words appear as contexts), and so $p_{context}(x) = p_{words}(x) = \frac{count(x)^\alpha}{|T|^\alpha}$. Setting $\alpha = 0.75$ gives rare noise words slightly higher probability (it dampens the probability for high occurrence words, e.g. *the*, and boosts one for rare words, e.g. *aardvark*).

So, after (6.9) our loss function is

$$
L = -\left[\sum_{(w,c)\in D} \log \sigma(v_c v_w) + \sum_{(w,c)\in D'} \log \sigma(-v_c v_w)\right]
$$

We minimize this loss function using stochastic gradient descent. Taking the derivative with respect to the different embeddings we obtain

$$
\begin{aligned}
\frac{\partial L}{\partial v_{c+}} &= -\sum_{(w,c)\in D} \frac{v_w}{1+e^{v_{c+} v_w}} \\
\frac{\partial L}{\partial v_{c-}} &= \sum_{(w,c)\in D'} \frac{v_w}{1+e^{-v_{c-} v_w}} \\
\frac{\partial L}{\partial v_w} &= -\sum_{(w,c+)\in D} \frac{v_{c+}}{1+e^{v_{c+} v_w}} + \sum_{(w,c-)\in D'} \frac{v_{c-}}{1+e^{-v_{c-} v_w}}
\end{aligned}
$$

The update equations going from time step t to $t+1$ in stochastic gradient descent are thus:

$$v_{c+}(t+1) = v_{c+}(t) - \eta \frac{\partial L}{\partial v_{c+}}(t)$$
$$v_{c-}(t+1) = v_{c-}(t) - \eta \frac{\partial L}{\partial v_{c-}}(t)$$
$$v_w(t+1) = v_w(t) - \eta \frac{\partial L}{\partial v_w}$$

Word2vec embeddings are static embeddings, meaning that the method learns one fixed embedding for each word in the vocabulary. This is in contrast with dynamic contextual embeddings like the popular BERT or ELMO representations, in which the vector for each word is different in different contexts.

6.7 GloVe

GloVe, coined from Global Vectors, is a model for distributed word representation. The model is an unsupervised learning algorithm for obtaining vector representations for words. This is achieved by mapping words into a meaningful space where the distance between words is related to semantic similarity. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. As log-bilinear regression model for unsupervised learning of word representations, it combines the features of two model families, namely the global matrix factorization and local context window methods.

The statistics of word occurrences in a corpus is the primary source of information available to all unsupervised methods for learning word representations. Let's consider the ratio of co-occurrence probabilities of 2 words i and j with various probe words k: P_{ik}/P_{jk}. For words k that are either related to both or to neither the ratio should be close to one. This argument suggests that the appropriate starting point for word vector learning should be with ratios of co-occurrence probabilities rather than the probabilities themselves. Noting that the ratio P_{ik}/P_{jk} depends on three words i, j, and k, the most general model takes the form,

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{6.10}$$

where $w \in \mathbb{R}^d$ are word vectors and $\tilde{w} \in \mathbb{R}^d$ are separate context word vectors. In this equation, the right-hand side is extracted from the corpus, and F may depend on some as-of-yet unspecified parameters. The number of possibilities for F is vast, but by enforcing a few desiderata we can select a unique choice. First, we would like F to encode the information which presents the ratio P_{ik}/P_{jk} in the word vector space. Since vector spaces are inherently linear structures, the most natural way to do this is with vector differences.

With this aim, we can restrict our consideration to those functions F that depend only on the difference of the two target words, modifying Eqn. (6.10) to,

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{6.11}$$

Next, we note that the arguments of F in Eqn. (6.11) are vectors while the right-hand side is a scalar. While F could be taken to be a complicated function parameterized by, e.g., a neural network, doing so would obfuscate the linear structure we are trying to capture. To avoid this issue, we can first take the dot product of the arguments,

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{P_{ik}}{P_{jk}} \tag{6.12}$$

which prevents F from mixing the vector dimensions in undesirable ways. Next, note that for word-word co-occurrence matrices, the distinction between a word and a context word is arbitrary and that we are free to exchange the two roles. To do so consistently, we must not only exchange $w \leftrightarrow \tilde{w}$ but also $X \leftrightarrow X^T$, where X is the matrix of word-word co-occurrence counts, whose entries X_{ij} tabulate the number of times word j occurs in the context of word i. Our final model should be invariant under this relabeling, but Eqn. (6.12) is not. However, the symmetry can be restored in two steps. First, we require that F be a homomorphism between the groups $(\mathbb{R}, +)$ and $(\mathbb{R}_{>0}, \times)$, i.e.,

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)} \tag{6.13}$$

which, by Eqn. (6.12), is solved by,

$$F(w_i^T \tilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i} \tag{6.14}$$

where $X_i = \sum_k X_{ik}$ is the number of times any word appears in the context of word i.

The solution to Eqn. (6.13) is $F = \exp$, or,

$$w_i^T \tilde{w}_k = \ln P_{ik} = \ln X_{ik} - \ln X_i \tag{6.15}$$

Next, we note that Eqn. (6.15) would exhibit the exchange symmetry if not for the $\ln X_i$ on the right-hand side. However, this term is independent of k so it can be absorbed into a bias b_i for w_i. Finally, adding an additional bias $\tilde{b}_k$ for $\tilde{w}_k$ restores the symmetry,

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \ln X_{ik} \tag{6.16}$$

Eqn. (6.16) is a drastic simplification over Eqn. (6.10), but it is actually ill-defined since the logarithm diverges whenever its argument is zero. One resolution to this issue is to include an additive shift in the logarithm, $\ln X_{ik} \rightarrow$

$\ln(X_{ik}+1)$, which maintains the sparsity of X while avoiding the divergences. The idea of factorizing the log of the co-occurrence matrix is closely related to LSA. A main drawback to this model is that it weighs all co-occurrences equally, even those that happen rarely or never. Such rare co-occurrences are noisy and carry less information than the more frequent ones - yet even just the zero entries account for 75-95% of the data in X, depending on the vocabulary size and corpus.

The following weighted least squares regression model addresses these problems. Casting Eqn. (6.16) as a least squares problem and introducing a weighting function $f(X_{ij})$ into the cost function gives us the model

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \ln X_{ij})^2 \tag{6.17}$$

where V is the size of the vocabulary. The weighting function should obey the following properties:

1. $f(0) = 0$. If f is viewed as a continuous function, it should vanish as $x \to 0$ fast enough that the $\lim_{x\to 0} f(x)\ln^2 x$ is finite.

2. $f(x)$ should be non-decreasing so that rare co-occurrences are not overweighted.

3. $f(x)$ should be relatively small for large values of x, so that frequent co-occurrences are not overweighted.

Of course a large number of functions satisfy these properties, but one class of functions that we found to work well can be parameterized as,

$$f(x) = \begin{cases} (x/x_{max})^\alpha & if\, x < x_{max} \\ 1 & otherwise \end{cases} \tag{6.18}$$

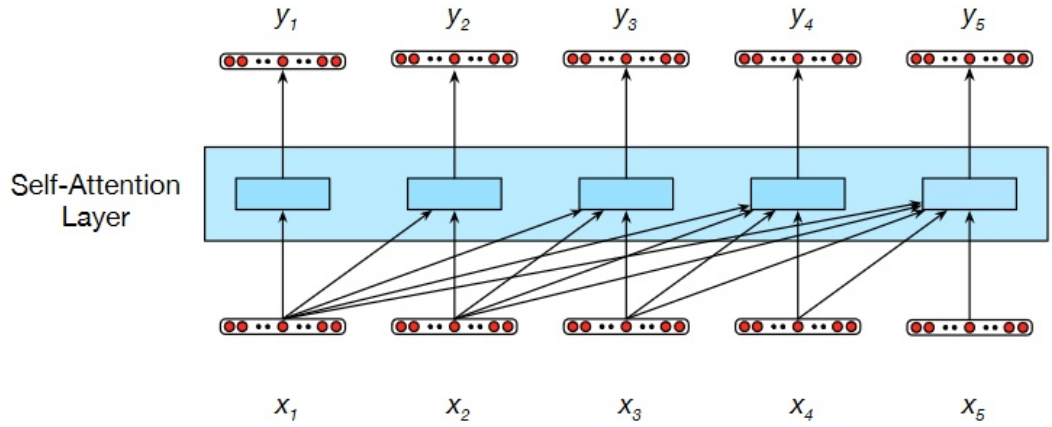
The performance of the model depends weakly on the cutoff, which we fix to $x_{max} = 100$ for all our experiments. We found that $\alpha = 3/4$ gives a modest improvement over a linear version with $\alpha = 1$.

6.8 Bidirectional Encoder Representations from Transformers (BERT)

Despite the ability of LSTMs to mitigate the loss of distant information due to the recurrence in RNNs, the underlying problem remains. Passing information forward through an extended series of recurrent connections leads to a loss of relevant information and to difficulties in training. Moreover, the inherently sequential nature of recurrent networks inhibits the use of parallel computational resources. These considerations led to the development of Transformers - an approach to sequence processing that eliminates recurrent connections and returns to architectures reminiscent of the fully connected networks.

Transformers map sequences of input vectors $(x_1, \ldots, x_n)$ to sequences of output vectors $(y_1, \ldots, y_n)$ of the same length. Transformers are made up of stacks of network layers consisting of simple linear layers, feedforward networks, and custom connections around them. In addition to these standard components, the key innovation of transformers is the use of self-attention layers. Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs.

A causal (or masked) self-attention model:



At the core of an attention-based approach is the ability to compare an item of interest to a collection of other items in way that reveals their relevance in the current context. It's essentially an auto-regression. Usually the dot product is used, but to allow for other possible comparisons, let's use a score function

$$score(x_i, x_j) = x_i x_j$$

then we normalize (to provide a probability distribution) to create weights

$$\begin{aligned} \alpha_{ij} &= softmax\left(score(x_i, x_j)\right) \ \forall j \leq i \\ &= \frac{exp\left(score(x_i, x_j)\right)}{\sum_{k=1}^{i} exp\left(score(x_i, x_k)\right)} \ \forall j \leq i \end{aligned} \tag{6.19}$$

and

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$

Unfortunately, this simple mechanism provides no opportunity for learning, everything is directly based on the original input values x. To allow for this, Transformers include additional parameters in the form of a set of weight matrices that operate over the input embeddings, called *query*, *key* and *value:*

$$q_i = W^Q x_i; \ k_i = W^K x_i; \ v_i = W^V x_i$$

Given input embeddings of size d_m, the dimensionality of these matrices are $d_q \times d_m$, $d_k \times d_m$ *and* $d_v \times d_m$, respectively. In the original Transformer work ([11]), d_m was 1024 and 64 for d_k, d_q and d_v. Given these projections,

$$score(x_i, x_j) = q_i k_j$$

and

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j$$

To avoid overflow in exponentiation in (6.19) the dot product needs to be scaled, typically

$$score(x_i, x_j) = \frac{q_i k_j}{\sqrt{d_k}}$$

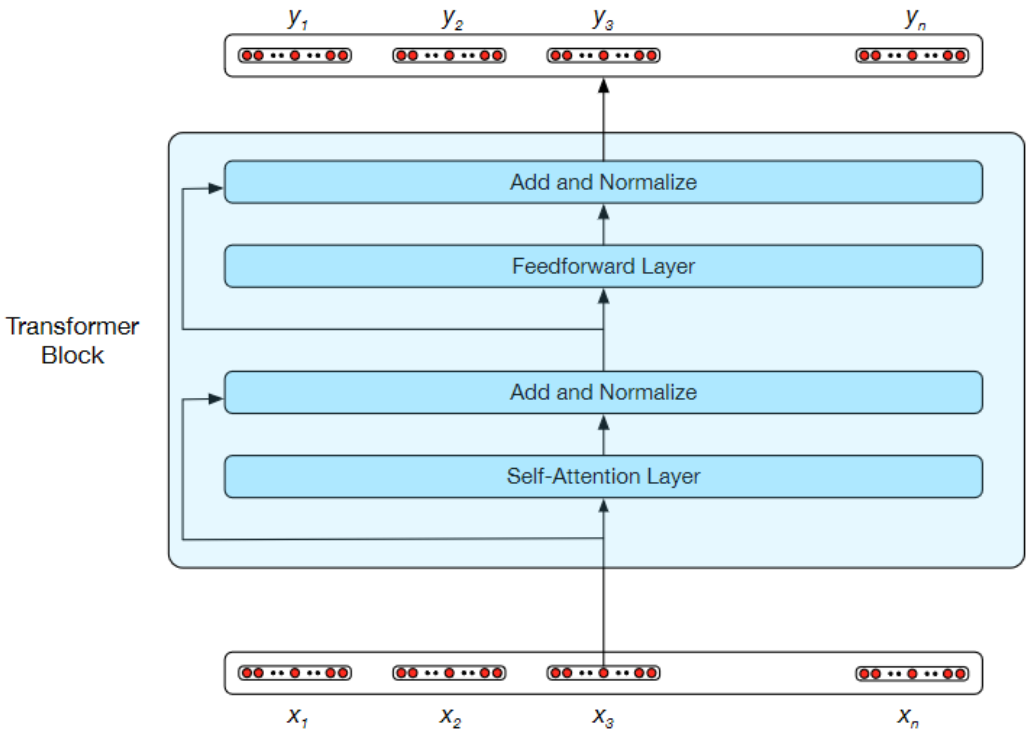
Since each output, y_i, is computed independently this entire process can be parallelized by taking advantage of efficient matrix multiplication routines by packing the input embeddings into a single matrix and multiplying it by the key, query and value matrices to produce matrices containing all the key, query and value vectors.

$$Q = W^Q X;\ K = W^K X;\ V = W^V X$$

and

$$SelfAttention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

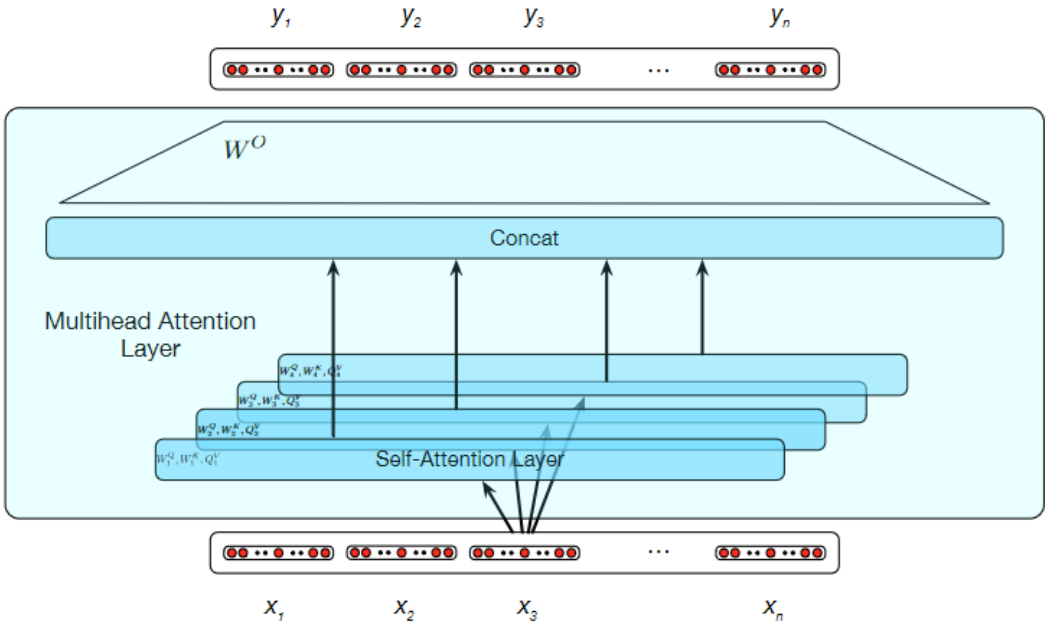
To exclude values following the query, set the upper triangle to $-\infty$.
A typical transformer block:



Multihead self-attention layers are sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters.

$$head_i = SelfAttention(W_i^Q X, W_i^K X, W_i^V X)$$
$$MultiHeadAttn(Q, K, V) = W^O(head_1 \oplus head_2 \ldots \oplus head_h)$$

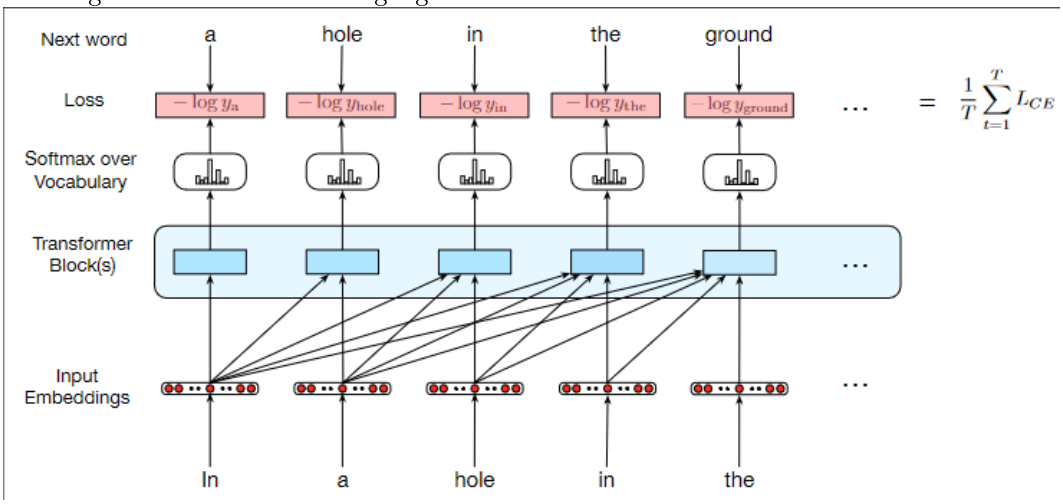
where $\oplus$ is concatenation.



The rest of the Transformer block with its feedforward layer, residual connections, and layer norms remains the same.

Unlike in RNN the positional information has to be encoded explicitly here.

Training a Transformer as a language model:



BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. There are two existing strategies for applying pre-trained language representations to down-stream tasks: feature-based and fine-tuning. The feature-based approach, such as ELMo, uses task-specific architectures that include the pre-

trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning all pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

A "masked language model" randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based only on its context.

Gaussian Error Linear Unit (GELU) is used which is

$$\frac{1}{2}x\left(1+erf\left(\frac{x}{\sqrt{2}}\right)\right)=x\Phi(x)$$

6.9 Transformer Embedding Dialogue (TED)

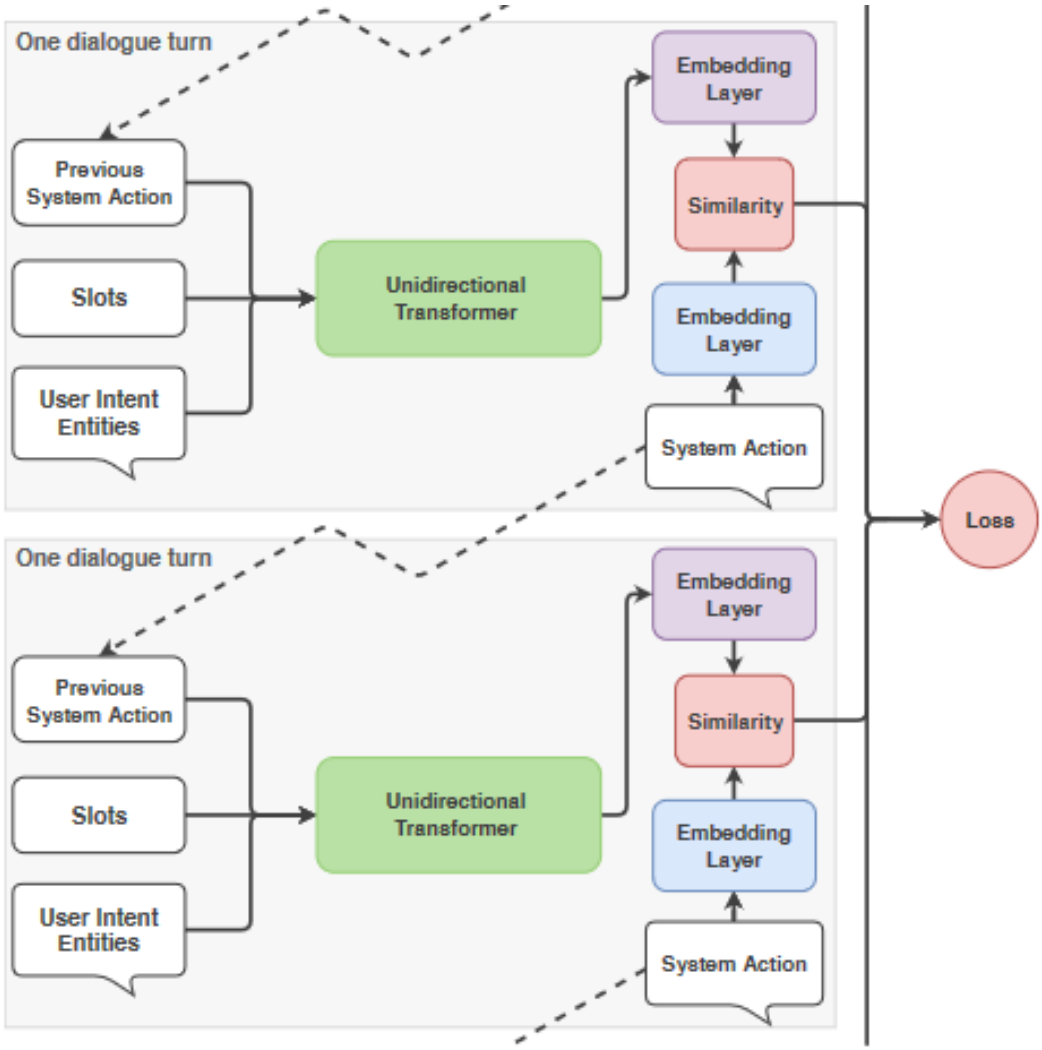


FIG. 1. A schematic representation of two time steps of the transformer embedding dialogue policy.

The transformer output a_{dialog} and system actions y_{action} are embedded into a single semantic vector space $h_{dialog} = E(a_{dialog})$, $h_{action} = E(y_{action})$, where $h \in \mathbb{R}^{20}$. The dot product loss is used to maximize the similarity $S^+ = h^T_{dialog} h^T_{action}$ with the target label y^+_{action} and minimize similarities $S^- = h^T_{dialog} h^-_{action}$ with negative samples y^-_{action}. Thus, the loss function for one dialogue reads

$$L_{dialog} = -\frac{1}{N}\sum^{N}\left[S^{+} - \log\left(e^{S^{+}} + \sum_{\Omega^{-}} e^{S^{-}}\right)\right]$$

where the second sum is taken over the set of negative samples Ω^{-} and the average is taken over time steps inside one dialogue.

The global loss is an average of all loss functions from all dialogues.

At inference time, the dot-product similarity serves as a ranker for the next utterance retrieval problem.

During modular training, we use a balanced batching strategy to mitigate class imbalance, as some system actions are far more frequent than others.

6.10 Dual Intent and Entity Transformer (DIET)

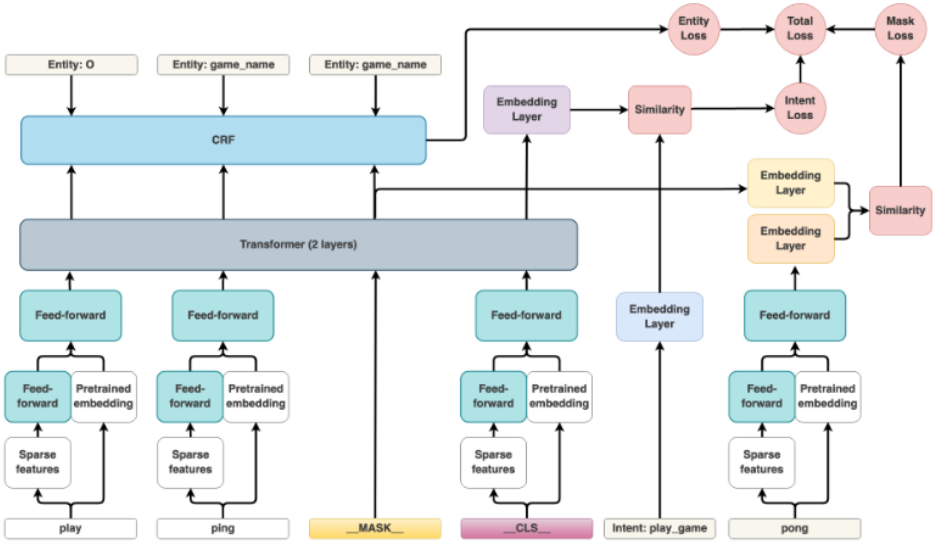


Figure 1: A schematic representation of the DIET architecture. The phrase "play ping pong" has the intent `play_game` and entity `game_name` with value "ping pong". Weights of the feed-forward layers are shared across tokens.

Input sentences are treated as a sequence of tokens, which can be either words or sub-words depending on the featurization pipeline. A special classification token__CLS__ is added to the end of each sentence. Each input token is featurized with what we call sparse features and/or dense features. Sparse features are token level one-hot encodings and multi-hot encodings of character n-grams (n≤5). Character n-grams contain a lot of redundant information, so to avoid overfitting we apply dropout to these sparse features. Dense features can be any pre-trained word embeddings: ConveRT, BERT or GloVe. Sparse features are passed through a fully connected layer with shared weights across all sequence steps to match the dimension of the dense features. The output of the

fully connected layer is concatenated with the dense features from pre-trained models.

For intent classification the transformer output for__CLS__ token a_{CLS} and intent labels y_{intent} are embedded into a single semantic vector space $h_{CLS} = E(a_{CLS})$, $h_{intent} = E(y_{intent})$, where $h \in \mathbb{R}^{20}$. The dot product loss is used to maximize the similarity $S^+ = h_{CLS}^T h_{intent}^+$ with the target label y_{intent}^+ and minimize similarities $S^- = h_{CLS}^T h_{intent}^-$ with negative samples y_{intent}^-. Thus, the loss function

$$L = -\frac{1}{N} \sum^{N} \left[S^+ - \log \left(e^{S^+} + \sum_{\Omega^-} e^{S^-} \right) \right]$$

where the second sum is taken over the set of negative samples Ω^- and the average is taken over all N examples.

At inference time, the dot-product similarity serves as a ranker over all possible intent labels.

Masking is added as an additional training objective to predict randomly masked input tokens.

The total loss is

$$L_{total} = L_{intent} + L_{entity} + L_{mask}$$

6.11 Intent / Entity identification

	intent	entity
Bag-of-words model with tf.idf	✓	
Naive Bayes	✓	
CRF		✓
GloVe	✓	✓
BERT	✓	✓
TED	✓	✓
DIET	✓	✓

Part III
Neural Networks

7 Multilayer perceptron

Models for regression and classification comprise linear combinations of fixed basis functions. Such models have useful analytical and computational properties but that their practical applicability is limited by the curse of dimensionality.

In order to apply such models to large scale problems, it is necessary to adapt the basis functions to the data.

Support vector machines (SVMs) address this by first defining basis functions that are centered on the training data points and then selecting a subset of these during training. One advantage of SVMs is that, although the training involves nonlinear optimization, the objective function is convex, and so the solution of the optimization problem is relatively straightforward. The number of basis functions in the resulting models is generally much smaller than the number of training points, although it is often still relatively large and typically increases with the size of the training set. The relevance vector machine also chooses a subset from a fixed set of basis functions and typically results in much sparser models. Unlike the SVM it also produces probabilistic outputs, although this is at the expense of a nonconvex optimization during training.

An alternative approach is to fix the number of basis functions in advance but allow them to be adaptive, in other words to use parametric forms for the basis functions in which the parameter values are adapted during training. The most successful model of this type in the context of pattern recognition is the feed-forward neural network, also known as the multilayer perceptron. In fact, "multilayer perceptron" is really a misnomer, because the model comprises multiple layers of logistic regression models (with continuous nonlinearities) rather than multiple perceptrons (with discontinuous nonlinearities). For many applications, the resulting model can be significantly more compact, and hence faster to evaluate, than a support vector machine having the same generalization performance. The price to be paid for this compactness, as with the relevance vector machine, is that the likelihood function, which forms the basis for network training, is no longer a convex function of the model parameters. In practice, however, it is often worth investing substantial computational resources during the training phase in order to obtain a compact model that is fast at processing new data.

The linear models for regression and classification are based on linear combinations of fixed nonlinear basis functions $\varphi_j(x)$ and take the form

$$y(x, w) = f\left(\sum_{j=1}^{M} w_j \varphi_j(x)\right) \tag{7.1}$$

where f is a nonlinear activation function in the case of classification and is the identity in the case of regression. Our goal is to extend this model by making the basis functions $\varphi_j(x)$ depend on parameters and then to allow these parameters to be adjusted, along with the coefficients w_j, during training. There are, of course, many ways to construct parametric nonlinear basis functions. Neural networks use basis functions that follow the same form as (7.1), so that each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters.

This leads to the basic neural network model, which can be described a series of functional transformations. First we construct M linear combinations of the input variables $x_1, \ldots, x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{7.2}$$

where $j = 1, \ldots, M$, and the superscript (1) indicates that the corresponding parameters are in the first "layer" of the network. The quantities a_j are known as activations. Each of them is then transformed using a differentiable, nonlinear activation function $h(\cdot)$ to give

$$z_j = h(a_j) \tag{7.3}$$

These quantities correspond to the outputs of the basis functions in (7.1) that, in the context of neural networks, are called hidden units. The nonlinear functions $h(\cdot)$ are generally chosen to be sigmoidal functions such as the logistic sigmoid or the *tanh* function. Following (7.1), these values are again linearly combined to give output unit activations

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{7.4}$$

where $k = 1, \ldots, K$, and K is the total number of outputs. We continue this process until the last layer where

$$y_k = \sigma(a_k) \tag{7.5}$$

To find the parameters w we can use the same sum-of-squares error function as in the linear regression. If we interpret the network as a conditional distribution then it corresponds to the maximum likelihood function.

In most cases there is no hope of finding an analytical solution to the equation $\nabla E(w) = 0$, so we resort to iterative numerical procedures, usually in the form

$$w(t+1) = w(t) + \Delta w(t)$$

The simplest approach to using gradient information is to choose the weight update to comprise a small step in the direction of the negative gradient, so that

$$w(t+1) = w(t) - \eta \nabla E\left(w(t)\right) \tag{7.6}$$

where the parameter $\eta > 0$ is the learning rate. This is the gradient descent optimization. After each such update, the gradient is re-evaluated for the new weight vector and the process repeated. Note that the error function is defined with respect to a training set, and so each step requires that the entire training set be processed in order to evaluate ∇E. Techniques that use the whole data set at once are called batch methods.

There is, however, an on-line version of gradient descent that has proved useful in practice for training neural networks on large data sets. Error functions

based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point

$$E(w) = \sum_{n=1}^{N} E_n(w)$$

On-line gradient descent, also known as sequential gradient descent or stochastic gradient descent, makes an update to the weight vector based on one data point at a time, so that

$$w(t+1) = w(t) - \eta \nabla E_n\left(w(t)\right) \tag{7.7}$$

This update is repeated by cycling through the data either in sequence or by selecting points at random with replacement. There are of course intermediate scenarios in which the updates are based on batches of data points.

A property of on-line gradient descent is the possibility of escaping from local minima, since a stationary point with respect to the error function for the whole data set will generally not be a stationary point for each data point individually.

One way to control the complexity of a neural network model in order to avoid over-fitting is regularization with the simplest one being the quadratic (a.k.a. *ridge regression*), giving a regularized error of the form

$$\tilde{E}(w) = E(w) + \frac{\lambda}{2} w^T w \tag{7.8}$$

This regularizer can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector w.

If we perform a linear transformation on the input and output data then we get 2 equivalent networks (with linearly adjusted weights), if not for (7.8). An invariant regularizer is given by

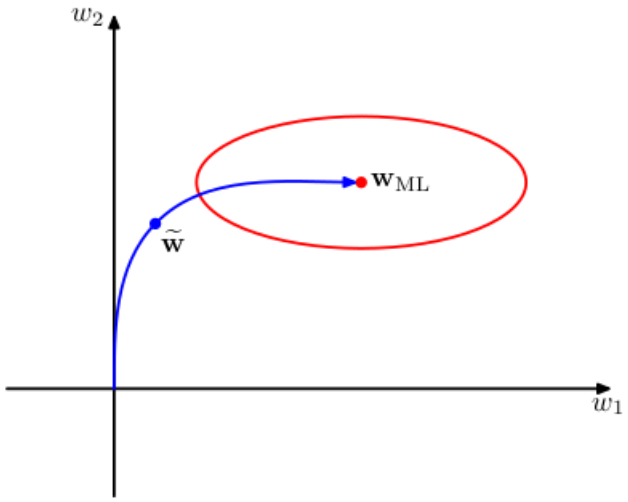
$$\frac{\lambda_1}{2} \sum_{w \in W_1} w^2 + \frac{\lambda_2}{2} \sum_{w \in W_2} w^2 \tag{7.9}$$

where W_1 and W_2 denote the set of weights in the first and second layers, respectively, and biases are excluded from the summations. This regularizer will remain unchanged under the linear weight transformations provided the regularization parameters are re-scaled using $\lambda_1 \to a^{1/2}\lambda_1$ and $\lambda_2 \to c^{-1/2}\lambda_2$.

The regularizer (7.9) corresponds to a prior of the form

$$p(w|\alpha_1, \alpha_2) \propto exp\left(-\frac{\alpha_1}{2} \sum_{w \in W_1} w^2 - \frac{\alpha_2}{2} \sum_{w \in W_2} w^2\right) \tag{7.10}$$

An alternative to regularization as a way of controlling the effective complexity of a network is the procedure of early stopping.



A schematic illustration of why early stopping can give similar results to weight decay in the case of a quadratic error function. The ellipse shows a contour of constant error, and $\mathbf{w}_{\mathrm{ML}}$ denotes the minimum of the error function. If the weight vector starts at the origin and moves according to the local negative gradient direction, then it will follow the path shown by the curve. By stopping training early, a weight vector $\widetilde{\mathbf{w}}$ is found that is qualitatively similar to that obtained with a simple weight-decay regularizer and training to the minimum of the regularized error, as

where the axes in weight space have been rotated to be parallel to the eigenvectors of the Hessian matrix.

Invariances: either add a lot of data or

1. A regularization term is added to the error function that penalizes changes in the model output when the input is transformed. This leads to the technique of *tangent propagation*. Consider the effect of a transformation on a particular input vector $\boldsymbol{x}_n$. Provided the transformation is continuous (such as translation or rotation, but not mirror reflection for instance), then the transformed pattern will sweep out a manifold $\mathcal{M}$ within the D-dimensional input space. Suppose the transformation is governed by a single parameter ξ (which might be rotation angle for instance). Then the subspace $\mathcal{M}$ swept out by $\boldsymbol{x}_n$ will be one-dimensional, and will be parameterized by ξ. Let the vector that results from acting on $\boldsymbol{x}_n$ by this transformation be denoted by $\boldsymbol{s}(\boldsymbol{x}_n, \xi)$, which is defined so that $\boldsymbol{s}(\boldsymbol{x}, 0) = \boldsymbol{x}$. Then the tangent to the curve $\mathcal{M}$ is given by the directional derivative $\boldsymbol{\tau} = \partial \boldsymbol{s}/\partial \xi$, and the tangent vector at the point $\boldsymbol{x}_n$ is given by

$$\boldsymbol{\tau}_n = \left.\frac{\partial \boldsymbol{s}(\boldsymbol{x}_n, \xi)}{\partial \xi}\right|_{\xi=0}$$

Under a transformation of the input vector, the network output vector will, in general, change. The derivative of output k with respect to ξ is given by

$$\left.\frac{\partial y_k}{\partial \xi}\right|_{\xi=0} = \sum_{i=1}^{D} \frac{\partial y_k}{\partial x_i} \left.\frac{\partial x_i}{\partial \xi}\right|_{\xi=0} = \sum_{i=1}^{D} J_{ki}\tau_i$$

where J_{ki} is the (k, i) element of the Jacobian matrix J. This result can be used to modify the standard error function, so as to encourage local

invariance in the neighborhood of the data points, by the addition to the original error function E of a regularization function Ω to give a total error function of the form

$$\tilde{E} = E + \lambda\Omega$$

where λ is a regularization coefficient and

$$\Omega = \frac{1}{2}\sum_n \sum_k \left(\frac{\partial y_{nk}}{\partial \xi}\bigg|_{\xi=0} \right)^2 = \frac{1}{2}\sum_n \sum_k \left(\sum_{i=1}^{D} J_{nki}\tau_{ni} \right)^2 \tag{7.11}$$

The regularization function will be zero when the network mapping function is invariant under the transformation in the neighborhood of each pattern vector, and the value of the parameter λ determines the balance between fitting the training data and learning the invariance property.

2. Invariance is built into the pre-processing by extracting features that are invariant under the required transformations.

3. The final option is to build the invariance properties into the structure of a neural network (or into the definition of a kernel function in the case of techniques such as the relevance vector machine). One way to achieve this is through the use of local receptive fields, shared weights, local convolutions, and subsampling. **Convolutional neural nets** learn the kernel via data.

We have seen that one way to encourage invariance of a model to a set of transformations is to expand the training set using transformed versions of the original input patterns. Here we show that this approach is closely related to the technique of tangent propagation.

We shall consider a transformation governed by a single parameter ξ and described by the function $\boldsymbol{s}(\boldsymbol{x}_n, \xi)$, with $\boldsymbol{s}(\boldsymbol{x}, 0) = \boldsymbol{x}$. We shall also consider a sum-of-squares error function. The error function for untransformed inputs can be written (in the infinite data set limit) in the form

$$E = \frac{1}{2}\int\int [y(\boldsymbol{x}) - t]^2 \, p(t|\boldsymbol{x})p(\boldsymbol{x})d\boldsymbol{x}dt$$

Here we have considered a network having a single output, in order to keep the notation uncluttered. If we now consider an infinite number of copies of each data point, each of which is perturbed by the transformation in which the parameter ξ is drawn from a distribution $p(\xi)$, then the error function defined over this expanded data set can be written as

$$\tilde{E} = \frac{1}{2}\int\int\int [y(\boldsymbol{s}(\boldsymbol{x}, \xi)) - t]^2 \, p(t|\boldsymbol{x})p(\boldsymbol{x})p(\xi)d\boldsymbol{x}dtd\xi \tag{7.12}$$

We now assume that the distribution $p(\xi)$ has zero mean with small variance, so that we are only considering small transformations of the original input

vectors. We can then expand the transformation function as a Taylor series in powers of ξ to give

$$\begin{aligned}\boldsymbol{s}(\boldsymbol{x},\xi) &= \boldsymbol{s}(\boldsymbol{x},0) + \xi\frac{\partial}{\partial\xi}\boldsymbol{s}(\boldsymbol{x},\xi)\bigg|_{\xi=0} + \frac{\xi^2}{2}\frac{\partial^2}{\partial\xi^2}\boldsymbol{s}(\boldsymbol{x},\xi)\bigg|_{\xi=0} + O(\xi^3) \\ &= \boldsymbol{x} + \xi\boldsymbol{\tau} + \frac{1}{2}\xi^2\boldsymbol{\tau}' + O(\xi^3)\end{aligned}$$

where $\boldsymbol{\tau}'$ denotes the second derivative of $\boldsymbol{s}(\boldsymbol{x},\xi)$ with respect to ξ evaluated at $\xi = 0$. This allows us to expand the model function to give

$$y(\boldsymbol{s}(\boldsymbol{x},\xi)) = y(\boldsymbol{x}) + \xi\boldsymbol{\tau}^T\nabla y(\boldsymbol{x}) + \frac{\xi^2}{2}\left[(\boldsymbol{\tau}')^T\nabla y(\boldsymbol{x}) + \boldsymbol{\tau}^T\nabla\nabla y(\boldsymbol{x})\boldsymbol{\tau}\right] + O(\xi^3)$$

Substituting into the mean error function (7.12) and expanding, we then have

$$\begin{aligned}\tilde{E} = &\frac{1}{2}\int\int \left[y(\boldsymbol{x}) - t\right]^2 p(t|\boldsymbol{x})p(\boldsymbol{x})d\boldsymbol{x}dt \\ &+ \mathbb{E}\left[\xi\right]\int\int \left[y(\boldsymbol{x}) - t\right]\boldsymbol{\tau}^T\nabla y(\boldsymbol{x})p(t|\boldsymbol{x})p(\boldsymbol{x})d\boldsymbol{x}dt \\ &+ \mathbb{E}\left[\xi^2\right]\int\int\left\{\left[y(\boldsymbol{x}) - t\right]\frac{1}{2}\left[(\boldsymbol{\tau}')^T\nabla y(\boldsymbol{x}) + \boldsymbol{\tau}^T\nabla\nabla y(\boldsymbol{x})\boldsymbol{\tau}\right]\right. \\ &\left.+ \left(\boldsymbol{\tau}^T\nabla y(\boldsymbol{x})\right)^2\right\}p(t|\boldsymbol{x})p(\boldsymbol{x})d\boldsymbol{x}dt + O(\xi^3)\end{aligned}$$

Because the distribution of transformations has zero mean we have $\mathbb{E}\left[\xi\right] = 0$. Also, we shall denote $\mathbb{E}\left[\xi^2\right]$ by λ. Omitting terms of $O(\xi^3)$, the average error function then becomes

$$\tilde{E} = E + \lambda\Omega \tag{7.13}$$

where E is the original sum-of-squares error, and the regularization term Ω takes the form

$$\begin{aligned}\Omega = \int\left\{\left[y(\boldsymbol{x}) - \mathbb{E}\left[t|\boldsymbol{x}\right]\right]\frac{1}{2}\left[(\boldsymbol{\tau}')^T\nabla y(\boldsymbol{x}) + \boldsymbol{\tau}^T\nabla\nabla y(\boldsymbol{x})\boldsymbol{\tau}\right]\right. \\ \left.+ \left(\boldsymbol{\tau}^T\nabla y(\boldsymbol{x})\right)^2\right\}p(\boldsymbol{x})d\boldsymbol{x}\end{aligned}$$

in which we have performed the integration over t.

We can further simplify this regularization term as follows. The function that minimizes the sum-of-squares error is given by the conditional average $\mathbb{E}\left[t|\boldsymbol{x}\right]$ of the target values t. From (7.13) we see that the regularized error will equal the unregularized sum-of-squares plus terms which are $O(\xi)$, and so the network function that minimizes the total error will have the form

$$y(\boldsymbol{x}) = \mathbb{E}\left[t|\boldsymbol{x}\right] + O(\xi)$$

Thus, to leading order in ξ, the first term in the regularizer vanishes and we are left with

$$\Omega = \frac{1}{2} \int \left(\boldsymbol{\tau}^T \nabla y(\boldsymbol{x}) \right)^2 p(\boldsymbol{x}) d\boldsymbol{x} \tag{7.14}$$

which is equivalent to the tangent propagation regularizer (7.11).

If we consider the special case in which the transformation of the inputs simply consists of the addition of random noise, so that $\boldsymbol{x} \to \boldsymbol{x} + \xi$, then the regularizer takes the form

$$\Omega = \frac{1}{2} \int \left\| \nabla y(\boldsymbol{x}) \right\|^2 p(\boldsymbol{x}) d\boldsymbol{x} \tag{7.15}$$

which is known as *Tikhonov regularization*. Derivatives of this regularizer with respect to the network weights can be found using an extended backpropagation algorithm.

Many machine learning algorithms aim to overcome the curse of dimensionality by assuming that the data lies near a low-dimensional manifold. One of the early attempts to take advantage of the manifold hypothesis is the tangent distance algorithm. It is a non-parametric nearest neighbor algorithm in which the metric used is not the generic Euclidean distance but one that is derived from knowledge of the manifolds near which probability concentrates. It is assumed that we are trying to classify examples, and that examples on the same manifold share the same category. Since the classifier should be invariant to the local factors of variation that correspond to movement on the manifold, it would make sense to use as nearest neighbor distance between points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ the distance between the manifolds $\mathcal{M}_1$ and $\mathcal{M}_2$ to which they respectively belong. Although that may be computationally difficult (it would require solving an optimization problem, to find the nearest pair of points on $\mathcal{M}_1$ and $\mathcal{M}_2$), a cheap alternative that makes sense locally is to approximate $\mathcal{M}_i$ by its tangent plane at $\boldsymbol{x}_i$ and measure the distance between the two tangents, or between a tangent plane and a point. That can be achieved by solving a low-dimensional linear system (in the dimension of the manifolds). Of course, this algorithm requires one to specify the tangent vectors.

In a related spirit, the *tangent prop* algorithm trains a neural net classifier with an extra penalty to make each output $f(\boldsymbol{x})$ of the neural net locally invariant to known factors of variation. These factors of variation correspond to movement along the manifold near which examples of the same class concentrate. Local invariance is achieved by requiring $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ to be orthogonal to the known manifold tangent vectors $\boldsymbol{v}^{(i)}$ at $\boldsymbol{x}$, or equivalently that the directional derivative of f at $\boldsymbol{x}$ in the directions $\boldsymbol{v}^{(i)}$ be small by adding a regularization penalty Ω:

$$\Omega(f) = \sum_i \left((\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^T \boldsymbol{v}^{(i)} \right)^2 \tag{7.16}$$

This regularizer can of course be scaled by an appropriate hyper-parameter, and for most neural networks, we would need to sum over many outputs rather

than the lone output $f(\boldsymbol{x})$ described here for simplicity. As with the tangent distance algorithm, the tangent vectors are derived a priori, usually from the formal knowledge of the effect of transformations, such as translation, rotation, and scaling in images.

Autoencoders can estimate the manifold tangent vectors. These estimated tangent vectors go beyond the classical invariants that arise out of the geometry of images (such as translation, rotation, and scaling) and include factors that must be learned because they are object-specific (such as moving body parts). The algorithm proposed with the manifold tangent classifier is therefore simple: (1) use an autoencoder to learn the manifold structure by unsupervised learning, and (2) use these tangents to regularize a neural net classifier as in tangent prop (7.16).

In comparison to L^2 regularization, L^1 regularization results in a solution that is more sparse. This property has been used extensively as a *feature selection* mechanism. In particular, the well known *LASSO* (least absolute shrinkage and selection operator) model integrates an L^1 penalty with a linear model and a least-squares cost function. The L^1 penalty causes a subset of the weights to become zero, suggesting that the corresponding features may safely be discarded.

Many regularization strategies can be interpreted as MAP Bayesian inference, and that in particular, L^2 regularization is equivalent to MAP Bayesian inference with a Gaussian prior on the weights. For L^1 regularization, the penalty $\alpha\Omega(\boldsymbol{w}) = \alpha\sum_i |w_i|$ used to regularize a cost function is equivalent to the log-prior term that is maximized by MAP Bayesian inference when the prior is an isotropic Laplace distribution over $w \in \mathbb{R}^n$:

$$\ln p(\boldsymbol{w}) = \sum_i \ln \mathit{Laplace}(w_i; 0, \frac{1}{\alpha}) = -\alpha \left\|\boldsymbol{w}\right\|_1 + n \ln \alpha - n \ln 2$$

From the point of view of learning via maximization with respect to $\boldsymbol{w}$, we can ignore the $\ln \alpha - \ln 2$ terms because they do not depend on $\boldsymbol{w}$.

Consider the cost function regularized by a parameter norm penalty:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

We can minimize a function subject to constrains by constructing a generalized Lagrange function, consisting of the original objective function plus a set of penalties. Each penalty is a product between a coefficient, called a Karush-Kuhn-Tucker (KKT) multiplier, and a function representing whether the constraint is satisfied.

In some cases, regularization is necessary for machine learning problems to be properly defined. Many linear models in machine learning, including linear regression and PCA, depend on inverting the matrix $\boldsymbol{X}^T\boldsymbol{X}$. This is not possible when $\boldsymbol{X}^T\boldsymbol{X}$ is singular. This matrix can be singular whenever the data-generating distribution truly has no variance in some direction, or when

no variance is observed in some direction because there are fewer examples (rows of $\boldsymbol{X}$) than input features (columns of $\boldsymbol{X}$). In this case, many forms of regularization correspond to inverting $\boldsymbol{X}^T\boldsymbol{X} + \alpha\boldsymbol{I}$ instead. This regularized matrix is guaranteed to be invertible. That's the Moore-Penrose pseudoinverse:

$$\boldsymbol{X}^\dagger = \lim_{\alpha \to 0} (\boldsymbol{X}^T\boldsymbol{X} + \alpha\boldsymbol{I})^{-1}\boldsymbol{X}^T$$

Most datasets have some number of mistakes in the y labels. It can be harmful to maximize $\ln p(y|\boldsymbol{x})$ when y is a mistake. One way to prevent this is to explicitly model the noise on the labels. *Label smoothing* deals with this.

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function. We are not guaranteed, however, that the training algorithm will be able to learn that function.

In the paradigm of *semi-supervised learning*, both unlabeled examples from $P(\boldsymbol{x})$ and labeled examples from $P(\boldsymbol{x}, \boldsymbol{y})$ are used to estimate $P(\boldsymbol{y}|\boldsymbol{x})$ or predict $\boldsymbol{y}$ from $\boldsymbol{x}$. In the context of deep learning, semi-supervised learning usually refers to learning a representation $\boldsymbol{h} = f(\boldsymbol{x})$. The goal is to learn a representation so that examples from the same class have similar representations. *Unsupervised learning* can provide useful clues for how to group examples in representation space. Examples that cluster tightly in the input space should be mapped to similar representations.

Instead of having separate unsupervised and supervised components in the model, one can construct models in which a generative model of either $P(\boldsymbol{x})$ or $P(\boldsymbol{x}, \boldsymbol{y})$ shares parameters with a discriminative model of $P(\boldsymbol{y}|\boldsymbol{x})$. One can then trade off the supervised criterion $-\ln P(\boldsymbol{y}|\boldsymbol{x})$ with the unsupervised or generative one (such as $-\ln P(\boldsymbol{x})$ or $-\ln P(\boldsymbol{x}, \boldsymbol{y})$). The generative criterion then expresses a particular form of prior belief about the solution to the supervised learning problem, namely that the structure of $P(\boldsymbol{x})$ is connected to the structure of $P(\boldsymbol{y}|\boldsymbol{x})$ in a way that is captured by the shared parametrization. By controlling how much of the generative criterion is included in the total criterion, one can find a better trade-off than with a purely generative or a purely discriminative training criterion.

Multitask learning is a way to improve generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks.

From the point of view of deep learning, the underlying prior belief is the following: among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks.

Bagging (short for *bootstrap aggregating*) is a technique for reducing generalization error by combining several models. The idea is to train several different models separately, then have all the models vote on the output for test examples. This is an example of a general strategy in machine learning called *model averaging*. Techniques employing this strategy are known as *ensemble methods*.

The reason that model averaging works is that different models will usually not make all the same errors on the test set.

Dropout trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero.

Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent. In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

One can reduce the error rate on the original i.i.d. test set via *adversarial training* - training on adversarially perturbed examples from the training set. One of the primary causes of these adversarial examples is excessive linearity. Neural networks are built out of primarily linear building blocks. In some experiments the overall function they implement proves to be highly linear as a result. These linear functions are easy to optimize. Unfortunately, the value of a linear function can change very rapidly if it has numerous inputs. If we change each input by ϵ, then a linear function with weights $\boldsymbol{w}$ can change by as much as $\epsilon||\boldsymbol{w}||_1$, which can be a very large amount if $\boldsymbol{w}$ is high-dimensional. Adversarial training discourages this highly sensitive locally linear behavior by encouraging the network to be locally constant in the neighborhood of the training data. This can be seen as a way of explicitly introducing a local constancy prior into supervised neural nets.

Adversarial examples also provide a means of accomplishing semi-supervised learning. At a point $\boldsymbol{x}$ that is not associated with a label in the dataset, the model itself assigns some label $\hat{y}$. The model's label $\hat{y}$ may not be the true label, but if the model is high quality, then $\hat{y}$ has a high probability of providing the true label. We can seek an adversarial example $\hat{\boldsymbol{x}}$ that causes the classifier to output a label y' with $y' \neq \hat{y}$. Adversarial examples generated using not the true label but a label provided by a trained model are called *virtual adversarial examples*. The classifier may then be trained to assign the same label to $\boldsymbol{x}$ and $\boldsymbol{x}'$. This encourages the classifier to learn a function that is robust to small changes anywhere along the manifold where the unlabeled data lie. The assumption motivating this approach is that different classes usually lie on disconnected manifolds, and a small perturbation should not be able to jump from one class manifold to another class manifold.

Among the most widely used of these implicit "priors" is the smoothness prior, or local constancy prior. This prior states that the function we learn should not change very much within a small region, i.e. $f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon)$. If we have several good answers in some neighborhood, we would combine them (by some form of averaging or interpolation) to produce an answer that agrees with as many of them as much as possible.

While the k-nearest neighbors algorithm copies the output from nearby training examples, most kernel machines interpolate between training set outputs associated with nearby training examples. An important class of kernels is the family of local kernels, where $k(\boldsymbol{u}, \boldsymbol{v})$ is large when $\boldsymbol{u} = \boldsymbol{v}$ and decreases as $\boldsymbol{u}$ and $\boldsymbol{v}$ grow further apart from each other. A local kernel can be thought of as a similarity function that performs template matching, by measuring how closely a test example $\boldsymbol{x}$ resembles each training example $\boldsymbol{x}^{(i)}$. Much of the modern motivation for deep learning is derived from studying the limitations of local template matching and how deep models are able to succeed in cases where local template matching fails.

Decision trees also suffer from the limitations of exclusively smoothness-based learning, because they break the input space into as many regions as there are leaves and use a separate parameter (or sometimes many parameters for extensions of decision trees) in each region. If the target function requires a tree with at least n leaves to be represented accurately, then at least n training examples are required to fit the tree. A multiple of n is needed to achieve some level of statistical confidence in the predicted output.

Is there a way to represent a complex function that has many more regions to be distinguished than the number of training examples? Clearly, assuming only smoothness of the underlying function will not allow a learner to do that. For example, imagine that the target function is a kind of checkerboard. A checkerboard contains many variations, but there is a simple structure to them.

Other approaches to machine learning often make stronger, task-specific assumptions. For example, we could easily solve the checkerboard task by providing the assumption that the target function is periodic. Usually we do not include such strong, task-specific assumptions in neural networks so that they can generalize to a much wider variety of structures. AI tasks have structure that is much too complex to be limited to simple, manually specified properties such as periodicity, so we want learning algorithms that embody more general-purpose assumptions. The core idea in deep learning is that we assume that the data was generated by the *composition of factors*, or features, potentially at multiple levels in a hierarchy.

In practice, it is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm. Most of the advances in neural network learning over the past thirty years have been obtained by changing the model family rather than changing the optimization procedure. Stochastic gradient descent with momentum, which was used to train neural networks in the 1980s, remains in use in modern state-of-the-art neural network applications.

Continuation methods are also closely related to simulated annealing, which adds noise to the parameters.

Continuation methods traditionally were mostly designed with the goal of overcoming the challenge of local minima. Specifically, they were designed to reach a global minimum despite the presence of many local minima. To do so, these continuation methods would construct easier cost functions by "blurring"

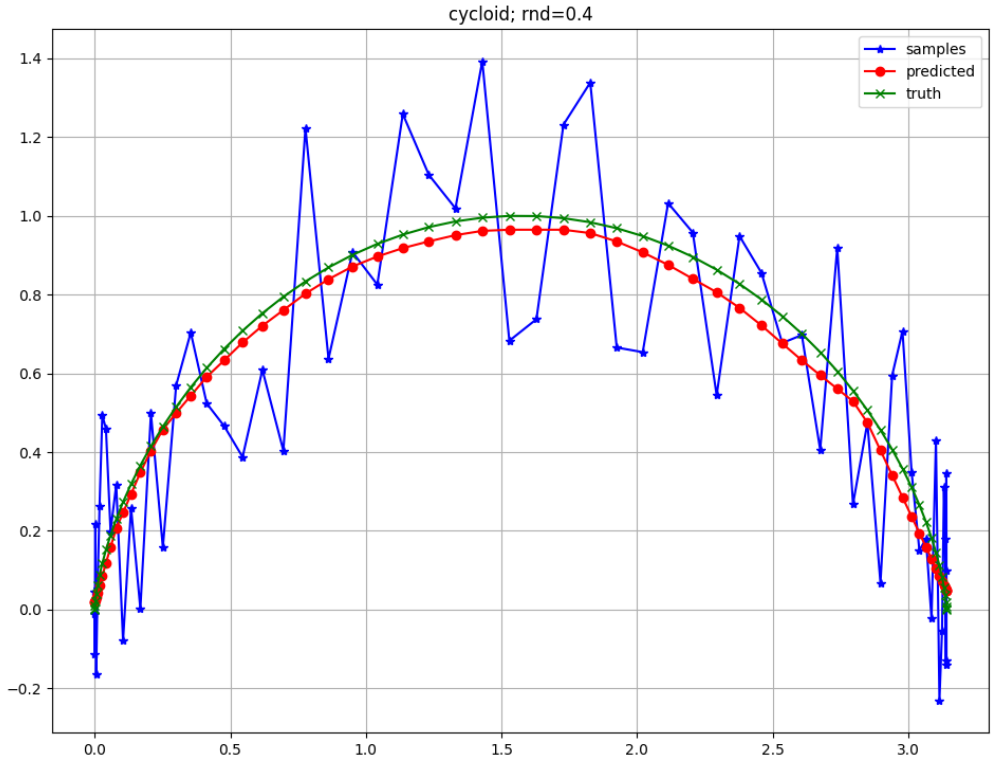
the original cost function. This blurring operation can be done by approximating

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\theta' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} \left[J(\boldsymbol{\theta}') \right]$$

via sampling. The intuition for this approach is that some nonconvex functions become approximately convex when blurred. In many cases, this blurring preserves enough information about the location of a global minimum that we can find the global minimum by solving progressively less-blurred versions of the problem.

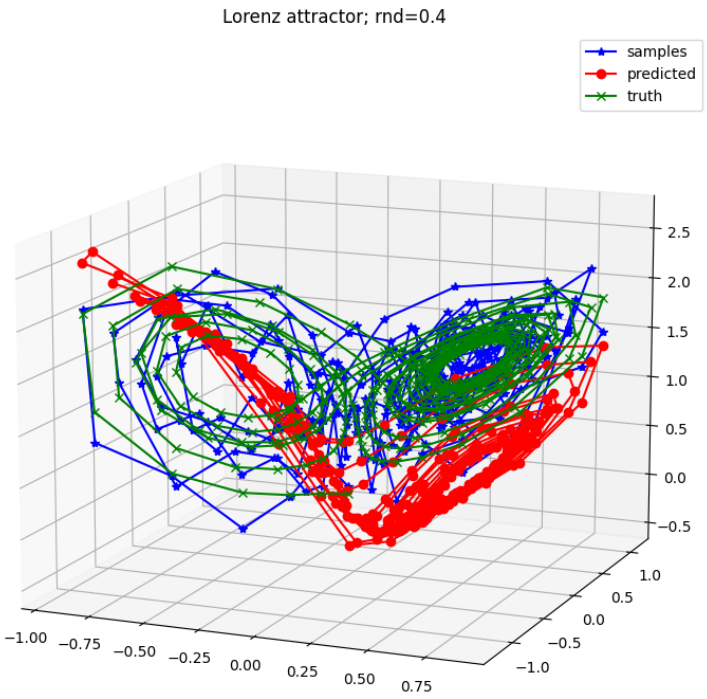
△ Here is an approximation of a cycloid with a 2 layer neural network with the ReLU activation functions. The cycloid is expressed via the parametric equations:

$$\begin{cases} x = r(t - \sin t) \\ y = r(1 - \cos t) \end{cases}$$



and similar for the Lorenz attractor:

$$\begin{cases} \frac{dx}{dt} = \sigma(y - x) \\ \frac{dy}{dt} = x(\rho - z) - y \\ \frac{dz}{dt} = xy - \beta z \end{cases}$$



▲

8 Convolutional Networks

Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant (to translations) representations.

The sparse interaction/connectivity is due to (much) smaller kernel size.

To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, that is, shifts it, then the convolution function is equivariant to g.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood,

the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel. In all cases, pooling helps to make the representation approximately invariant to small translations of the input.

9 Recurrent Neural Networks

In an RNN for an external signal $\boldsymbol{x}(t)$, the system state is described as

$$\boldsymbol{s}(t) = f\left[\boldsymbol{s}(t-1), \boldsymbol{x}(t); \boldsymbol{\theta}\right] \tag{9.1}$$

A variant that propagates hidden units in time represents a universal approximator in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size. For example, for the hyperbolic tangent activation function and the softmax operation

$$\begin{aligned}
\boldsymbol{a}(t) &= \boldsymbol{b} + W\boldsymbol{h}(t-1) + U\boldsymbol{x}(t) \\
\boldsymbol{h}(t) &= \tanh\left(\boldsymbol{a}(t)\right) \\
\boldsymbol{o}(t) &= \boldsymbol{c} + V\boldsymbol{h}(t) \\
\hat{\boldsymbol{y}}(t) &= softmax\left(\boldsymbol{o}(t)\right)
\end{aligned}$$

And the total loss for, e.g. the negative log-likelihood, is

$$L\left[\left(\boldsymbol{x}(1), \ldots, \boldsymbol{x}(\tau)\right), \left(\boldsymbol{y}(1), \ldots, \boldsymbol{y}(\tau)\right)\right] = -\sum_t \log p_{model}\left[y(t)\middle|\left(\boldsymbol{x}(1), \ldots, \boldsymbol{x}(t)\right)\right]$$

where $p_{model}\left[y(t)\middle|\left(\boldsymbol{x}(1), \ldots, \boldsymbol{x}(t)\right)\right]$ is given by reading the entry for $y(t)$ from the model's output vector $\hat{\boldsymbol{y}}(t)$.

A variant network with recurrent connections only from the output at one time step to the hidden units at the next time step is strictly less powerful. Because the output units are explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input, unless the user knows how to describe the full state of the system and provides it as part of the training set targets. However, the advantage is that training can be parallelized.

Recursive neural networks represent yet another generalization of recurrent networks with a computational graph structured as a deep tree instead of the chain-like structure of RNNs.

Like leaky units, **gated RNN**s are based on the idea of creating paths through time that have derivatives that neither vanish nor explode. Leaky units did this with connection weights that were either manually chosen constants or were parameters. Gated RNNs generalize this to connection weights that may change at each time step.

Leaky units allow the network to accumulate information (such as evidence for a particular feature or category) over a long duration. Once that information has been used, however, it might be useful for the neural network to forget the old state.

The clever idea of introducing self-loops to produce paths where the gradient can flow for long duration is a core contribution of the initial **long short-term memory** (LSTM) model. A crucial addition has been to make the weight on this self-loop conditioned on the context, rather than fixed. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, we mean that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself.

The self-loop weight (or the associated time constant) is controlled by a forget gate unit $f_i^{(t)}$ (for time step t and cell i) which sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

where $\boldsymbol{x}^{(t)}$ is the current input vector and $\boldsymbol{h}^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f, \boldsymbol{U}^f, \boldsymbol{W}^f$ are respectively biases, input weights, and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

where $\boldsymbol{b}, \boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights, and recurrent weights into the LSTM cell. The external input gate unit $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the output gate $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$$

which has parameters $\boldsymbol{b}^o, \boldsymbol{U}^o, \boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively.

Gated recurrent units, or GRUs, simultaneously control the forgetting factor and the decision to update the state unit. The update equations are the following:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + \left(1 - u_i^{(t-1)}\right) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right)$$

where $\boldsymbol{u}$ stands for "update" gate and $\boldsymbol{r}$ for "reset" gate. Their value is defined as usual:

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right)$$

and

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right)$$

The reset and update gates can individually "ignore" parts of the state vector.

The update gates act like conditional leaky integrators that can linearly gate any dimension, thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) by replacing it with the new "target state" value (toward which the leaky integrator wants to converge). The reset gates control which parts of the state get used to compute the next target state, introducing an additional nonlinear effect in the relationship between past state and future state.

Many more variants around this theme can be designed. For example the reset gate (or forget gate) output could be shared across multiple hidden units. Alternately, the product of a global gate (covering a whole group of units, such as an entire layer) and a local gate (per unit) could be used to combine global control and local control. Several investigations over architectural variations of the LSTM and GRU, however, found no variant that would clearly beat both of these across a wide range of tasks.

To prevent the exploding gradients, clipping the gradient technique is used.

To help with the vanishing gradients, the following regularizer can be used:

$$\Omega = \sum_t \left(\frac{\left\| (\nabla_{\boldsymbol{h}^{(t)}} L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \right\|}{\|\nabla_{\boldsymbol{h}^{(t)}} L\|} - 1 \right)^2$$

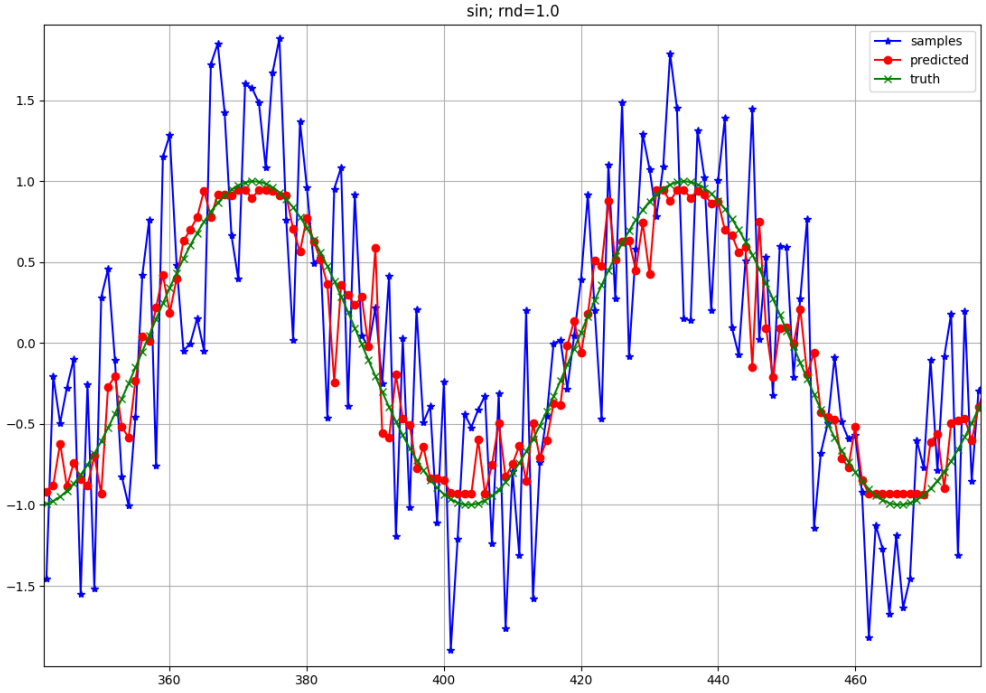
Computing the gradient of this regularizer may appear difficult, but an approximation can be used in which we consider the back-propagated vectors $\nabla_{\boldsymbol{h}^{(t)}} L$ as if they were constants (for the purpose of this regularizer, so that there is no need to back-propagate through them).

Model compression. Often one needs large models to learn some function $f(\boldsymbol{x})$, but do so using many more parameters than are necessary for the task. Their size is necessary only because of the limited number of training examples. As soon as we have fit this function $f(\boldsymbol{x})$, we can generate a training set containing infinitely many examples, simply by applying f to randomly sampled points $\boldsymbol{x}$, preferably from a distribution resembling the actual test inputs. We then train a new, smaller model to match $f(\boldsymbol{x})$ on these points.

△ Here is an extrapolation for the sine function using an RNN.

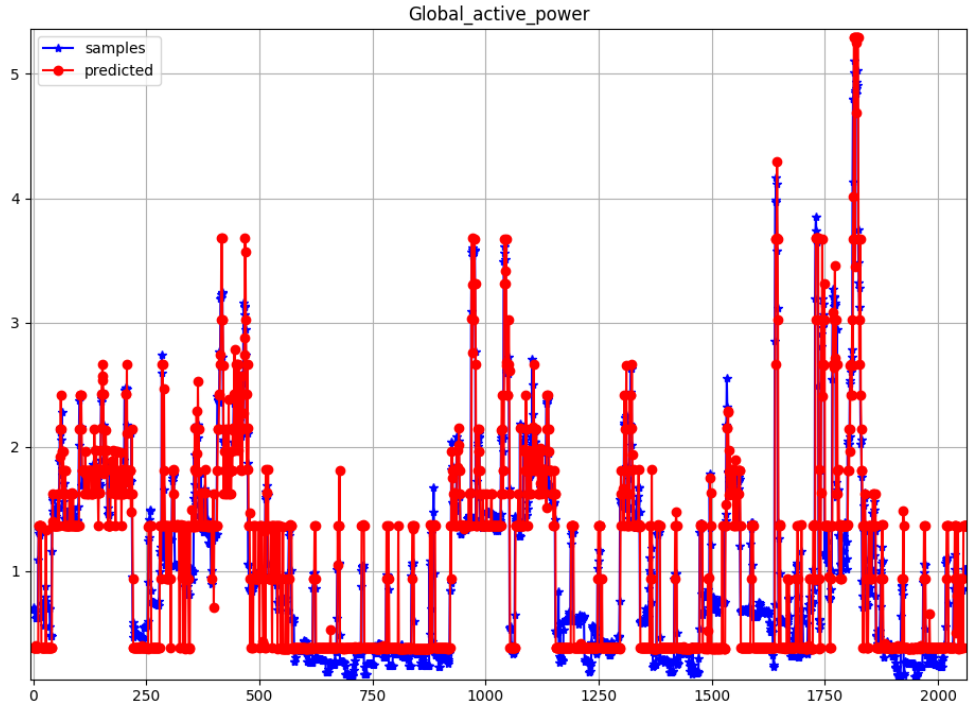
$$y = \sin x + \mathcal{U}$$

where $\mathcal{U}$ is a uniformly distributed noise.

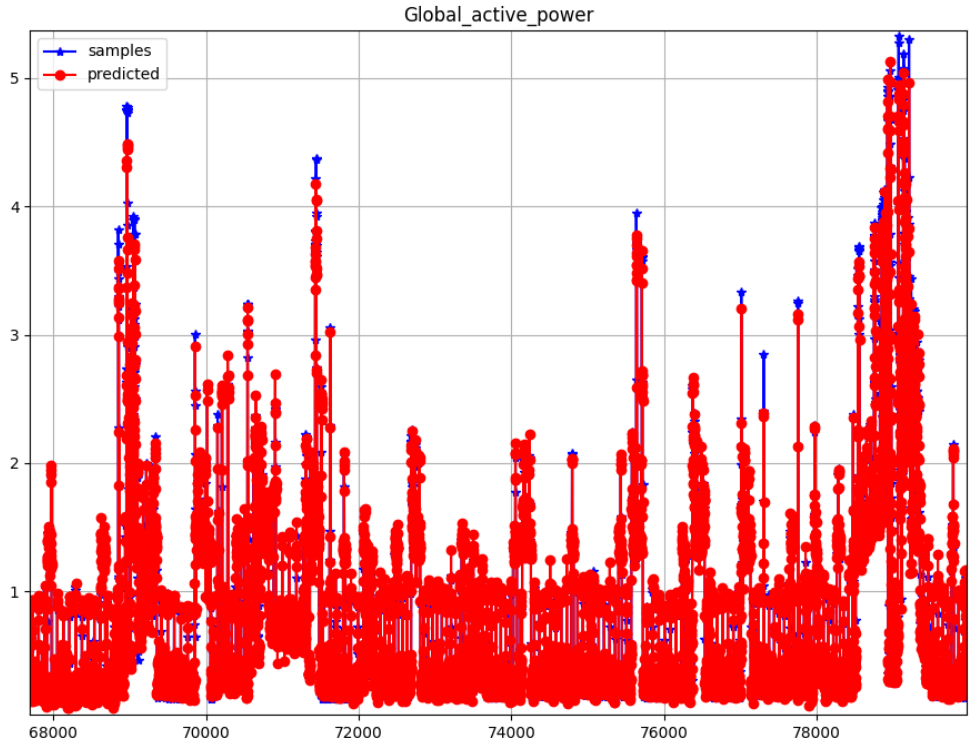


▲

△ Here is an example of prediction for electric power consumption using RNN on the household dataset (https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power



and the same using LSTM:



▲

10 Autoencoders

An autoencoder is a neural network that is trained to attempt to copy its input to its output. The network may be viewed as consisting of two parts: an encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder that produces a reconstruction $\boldsymbol{r} = g(\boldsymbol{h})$. If an autoencoder succeeds in simply learning to set $g(f(\boldsymbol{x})) = \boldsymbol{x}$ everywhere, then it is not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings $p_{encoder}(\boldsymbol{h}|\boldsymbol{x})$ and $p_{decoder}(\boldsymbol{x}|\boldsymbol{h})$.

When the decoder is linear and L is the mean squared error, an undercomplete (with less dimension) autoencoder learns to span the same subspace as PCA. In this case, an autoencoder trained to perform the copying task has

learned the principal subspace of the training data as a side effect. Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can thus learn a more powerful nonlinear generalization of PCA.

Autoencoders can be used to learn lower dimensional manifolds.

11 Representation Learning

The deep learning renaissance of 2006 began with the discovery that the greedy layer-wise unsupervised pretraining procedure could be used to find a good initialization for a joint learning procedure over all the layers, and that this approach could be used to successfully train even fully connected architectures. Today, we now know that greedy layer-wise pretraining is not required to train fully connected deep architectures, but the unsupervised pretraining approach was the first method to succeed.

Greedy layer-wise pretraining is called greedy because it is a greedy algorithm, meaning that it optimizes each piece of the solution independently, one piece at a time, rather than jointly optimizing all pieces.

Greedy layer-wise unsupervised pretraining protocol.
Given the following: Unsupervised feature learning algorithm $\mathcal{L}$, which takes a training set of examples and returns an encoder or feature function f. The raw input data is $\boldsymbol{X}$, with one row per example, and $f^{(1)}(\boldsymbol{X})$ is the output of the first stage encoder on $\boldsymbol{X}$. In the case where fine-tuning is performed, we use a learner $\mathcal{T}$, which takes an initial function f, input examples $\boldsymbol{X}$ (and in the supervised fine-tuning case, associated targets $\boldsymbol{Y}$), and returns a tuned function. The number of stages is m.
$f \leftarrow$ Identity function
$\tilde{\boldsymbol{X}} = \boldsymbol{X}$
for $k = 1, \ldots, m$ do
  $f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}})$
  $f \leftarrow f^{(k)} \circ f$
  $\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\boldsymbol{X})$
end for
if fine-tuning then
  $f \leftarrow \mathcal{T}(f, \boldsymbol{X}, \boldsymbol{Y})$
end if
return f

Here is a non-exhaustive list of generic regularization strategies. It represents some concrete examples of how learning algorithms can be encouraged to discover features that correspond to underlying factors.

- **Smoothness**: This is the assumption that $f(x + \epsilon d) \approx f(x)$ for unit d and small ϵ. This assumption allows the learner to generalize from training examples to nearby points in input space. Many machine learning

algorithms leverage this idea, but it is insufficient to overcome the curse of dimensionality.

- **Linearity**: Many learning algorithms assume that relationships between some variables are linear. This allows the algorithm to make predictions even very far from the observed data, but can sometimes lead to overly extreme predictions. Most simple machine learning algorithms that do not make the smoothness assumption instead make the linearity assumption. These are in fact different assumptions - linear functions with large weights applied to high-dimensional spaces may not be very smooth.

- **Multiple explanatory factors**: Many representation learning algorithms are motivated by the assumption that the data is generated by multiple underlying explanatory factors, and that most tasks can be solved easily given the state of each of these factors. This view motivates semi-supervised learning via representation learning. Learning the structure of $p(\boldsymbol{x})$ requires learning some of the same features that are useful for modeling $p(\boldsymbol{y}|\boldsymbol{x})$ because both refer to the same underlying explanatory factors. This view motivates the use of distributed representations, with separate directions in representation space corresponding to separate factors of variation.

- **Causal factors**: The model is constructed in such a way that it treats the factors of variation described by the learned representation $\boldsymbol{h}$ as the causes of the observed data $\boldsymbol{x}$, and not vice versa. This is advantageous for semi-supervised learning and makes the learned model more robust when the distribution over the underlying causes changes or when we use the model for a new task.

- **Depth**, or a **hierarchical organization of explanatory factors**: High-level, abstract concepts can be defied in terms of simple concepts, forming a hierarchy. From another point of view, the use of a deep architecture expresses our belief that the task should be accomplished via a multistep program, with each step referring back to the output of the processing accomplished via previous steps.

- **Shared factors across tasks**: When we have many tasks corresponding to different y_i variables sharing the same input $\boldsymbol{x}$, or when each task is associated with a subset or a function $f^{(i)}(\boldsymbol{x})$ of a global input $\boldsymbol{x}$, the assumption is that each y_i is associated with a different subset from a common pool of relevant factors $\boldsymbol{h}$. Because these subsets overlap, learning all the $P(y_i|\boldsymbol{x})$ via a shared intermediate representation $P(\boldsymbol{h}|\boldsymbol{x})$ allows sharing of statistical strength between the tasks.

- **Manifolds**: Probability mass concentrates, and the regions in which it concentrates are locally connected and occupy a tiny volume. In the continuous case, these regions can be approximated by low-dimensional manifolds with a much smaller dimensionality than the original space where

the data live. Many machine learning algorithms behave sensibly only on this manifold. Some machine learning algorithms, especially autoencoders, attempt to explicitly learn the structure of the manifold.

- **Natural clustering**: Many machine learning algorithms assume that each connected manifold in the input space may be assigned to a single class. The data may lie on many disconnected manifolds, but the class remains constant within each one of these. This assumption motivates a variety of learning algorithms, including tangent propagation, double backprop, the manifold tangent classifier and adversarial training.

- **Temporal and spatial coherence**: Slow feature analysis and related algorithms make the assumption that the most important explanatory factors change slowly over time, or at least that it is easier to predict the true underlying explanatory factors than to predict raw observations such as pixel values.

- **Sparsity**: Most features should presumably not be relevant to describing most inputs - there is no need to use a feature that detects elephant trunks when representing an image of a cat. It is therefore reasonable to impose a prior that any feature that can be interpreted as "present" or "absent" should be absent most of the time.

- **Simplicity of factor dependencies**: In good high-level representations, the factors are related to each other through simple dependencies. The simplest possible is marginal independence, $P(\boldsymbol{h}) = \prod_i P(\boldsymbol{h}_i)$, but linear dependencies or those captured by a shallow autoencoder are also reasonable assumptions. This can be seen in many laws of physics and is assumed when plugging a linear predictor or a factorized prior on top of a learned representation.

12 Structured Probabilistic Models for Deep Learning

One kind of structured probabilistic model is the directed graphical model (using DAG), otherwise known as the belief network or Bayesian network. Another popular language is that of undirected models, otherwise known as Markov random fields (MRFs) or Markov networks.

Context-specific independences are independences that are present dependent on the value of some variables in the network.

Gibbs sampling. Suppose we have a graphical model over an n-dimensional vector of random variables $\boldsymbol{x}$. We iteratively visit each variable x_i and draw a sample conditioned on all the other variables, from $p(x_i|x_{-i})$. Due to the separation properties of the graphical model, we can equivalently condition on only the neighbors of x_i. Unfortunately, after we have made one pass through the

graphical model and sampled all n variables, we still do not have a fair sample from $p(\boldsymbol{x})$. Instead, we must repeat the process and resample all n variables using the updated values of their neighbors. Asymptotically, after many repetitions, this process converges to sampling from the correct distribution. It can be difficult to determine when the samples have reached a sufficiently accurate approximation of the desired distribution.

13 Reinforcement Learning

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment.

Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards. This framework is intended to be a simple way of representing essential features of the artificial intelligence problem. These features include a sense of cause and effect, a sense of uncertainty and nondeterminism, and the existence of explicit goals.

The concepts of value and value function are key to most of the reinforcement learning methods. We take the position that value functions are important for efficient search in the space of policies. The use of value functions distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by evaluations of entire policies.

The online nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states.

The Bellman equation:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t | S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \quad for\, \forall\, s \in \mathcal{S} \end{aligned} \tag{13.1}$$

The optimal state-value function:

$$v_*(s) := \max_\pi v_\pi(s) \quad for\, \forall\, s \in \mathcal{S}$$

Hence, we can obtain π_* by greedy search among all actions at s_t and taking the action a which maximizes the following objective:

$$O = \mathbb{E}_{s_{t+1} \sim \mathcal{T}(s_{t+1}|s_t, a)}\left[v_{\pi_*}(s_{t+1})\right]$$

where $\mathcal{T}(s_{t+1}|s_t, a)$ is the so-called transition dynamics which construct a mapping of state-action pair at time t onto a set of states at time $t+1$. However, it is common sense that $\mathcal{T}$ is not available in RL settings. Therefore, the so-called Q-function: $q_\pi(s, a)$ is introduced as the alternative to $v_\pi(s)$:

$$q_\pi(s, a) = \mathbb{E}\left[G|s, a, \pi\right]$$

where the initial action a and the following policy π is pre-given. $q_\pi(s, a)$ denotes the expected return value of taking an action a in a state s following the policy π. Thus, given $q_\pi(s, a)$, the optimal policy π_* can be obtained by the greedy search among all actions and the corresponding $v_{\pi_*}(s)$ can be defined as:

$$v_{\pi_*}(s) = \max_a q_\pi(s, a)$$

This action-value function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s,\, A_t = a\right]$$

Iterative Policy Evaluation (in-place) for estimating $V \approx v_\pi$:

Input π, the policy to be evaluated.
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation.
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$.
loop:
  $\triangle \leftarrow 0$
  loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\triangle \leftarrow \max\left(\triangle, |v - V(s)|\right)$
until $\triangle < \theta$

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$:

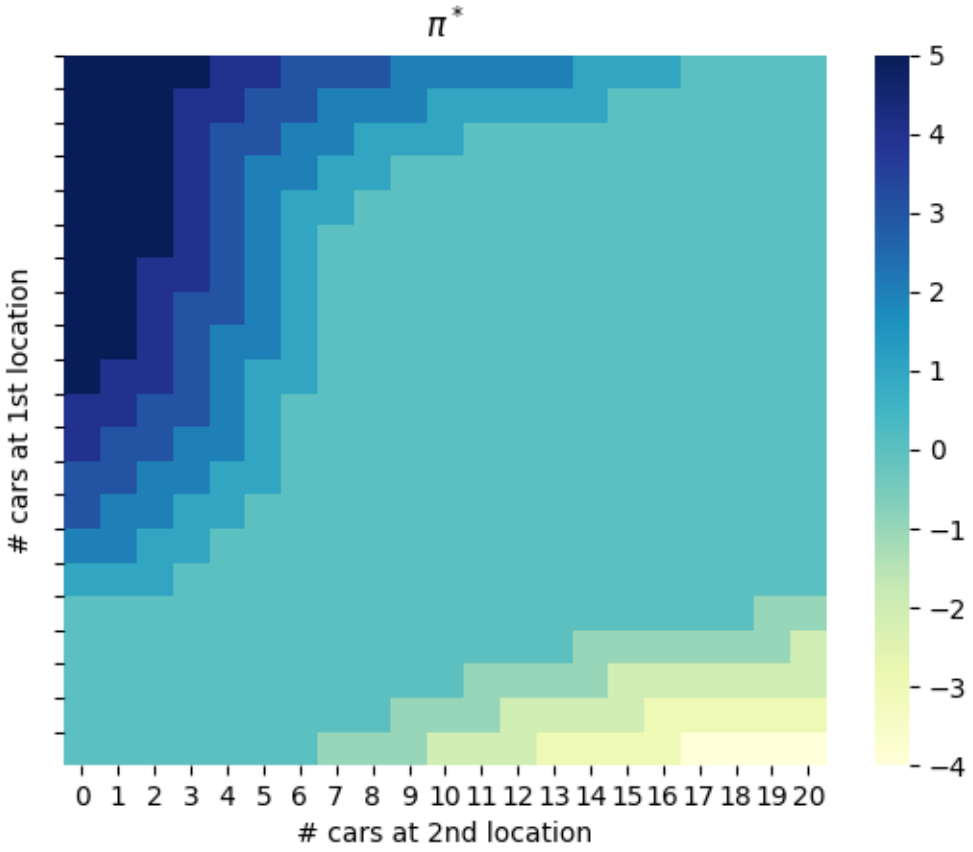
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation
   loop:
      $\triangle \leftarrow 0$
      loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s)) [r + \gamma V(s')]$
         $\triangle \leftarrow \max(\triangle, |v - V(s)|)$
   until $\triangle < \theta$ (a small positive number determining the accuracy of estimation)
3. Policy Improvement
   policy_stable$\leftarrow$ *true*
   for each $s \in \mathcal{S}$:
      old_action$\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a) [r + \gamma V(s')]$
      // if there are multiple policies that are equally good, select consistently, e.g. first
      if old_action $\neq \pi(s)$ then policy_stable $\leftarrow$ *false*
   if policy_stable then stop and return $V \approx v_*$ and $\pi \approx \pi_*$ else go to 2.

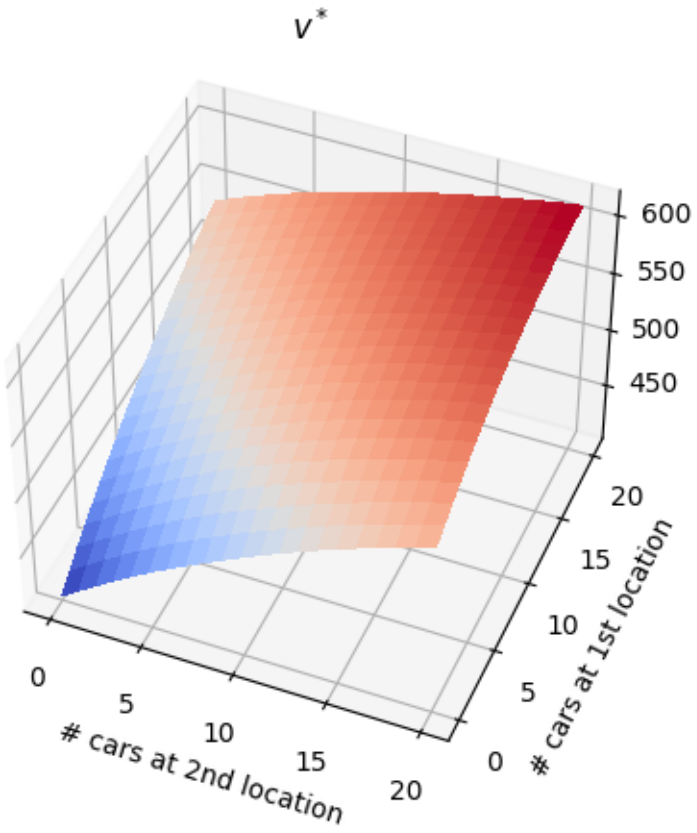
13.1 Car rental problem

$\triangle$

Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!} e^{-\lambda}$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight (negative numbers indicate transfers from the second location to the first).

The figures below show the optimal policy and optimal value function.


π^*
cars at 1st location
cars at 2nd location
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
5
4
3
2
1
0
−1
−2
−3
−4



▲

Value Iteration for estimating $\pi \approx \pi_*$:

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation.
Initialize $V(s)$ for all $s \in \mathcal{S}^+$ arbitrarily except that $V(terminal) = 0$,
where $\mathcal{S}^+$ is $\mathcal{S}$ plus the terminal state.
loop:
  $\triangle \leftarrow 0$
  loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\,[r + \gamma V(s')]$
    $\triangle \leftarrow \max(\triangle, |v - V(s)|)$
until $\triangle < \theta$
Output a deterministic policy $\pi \approx \pi_*$ such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\,[r + \gamma V(s')]$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement.

Dynamic Programming methods update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea bootstrapping. This is in a contrast to Monte Carlo sampling which uses independent estimates.

Planning requires a model of the environment. A distribution model consists of the probabilities of next states and rewards for possible actions; a sample model produces single transitions and rewards generated according to these probabilities. Dynamic programming requires a distribution model because it uses expected updates, which involve computing expectations over all the possible next states and rewards. A sample model, on the other hand, is what is needed to simulate interacting with the environment during which sample updates, like those used by many reinforcement learning algorithms, can be used. Sample models are generally much easier to obtain than distribution models.

13.2 Server access control problem

△

This is a decision task involving access control to a set of 10 servers. Customers of 4 different priorities arrive at a single queue. If given access to a server, the customers pay a reward of 1, 2, 4, or 8 to the server, depending on their priority, with higher priority customers paying more. In each time step, the customer at the head of the queue is either accepted (assigned to one of the servers) or rejected (removed from the queue, with a reward of zero). In either case, on the next time step the next customer in the queue is considered. The queue never empties, and the priorities of the customers in the queue are uniformly randomly distributed. Of course a customer cannot be served if there is no free server; the customer is always rejected in this case. Each busy server becomes free with probability $p = 0.06$ on each time step. Although we have just described them for definiteness, let us assume the statistics of arrivals and departures are unknown. The task is to decide on each step whether to accept or reject the next customer, on the basis of his priority and the number of free servers, so as to maximize long-term reward without discounting.

In the average-reward setting, the quality of a policy π is defined as the average rate of reward, or simply average reward, while following that policy, which we denote as $r(\pi)$:

$$\begin{aligned} r(\pi) &:= \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}\left[R_t | S_0, A_{0:t-1} \sim \pi\right] \\ &= \lim_{t\to\infty} \mathbb{E}\left[R_t | S_0, A_{0:t-1} \sim \pi\right] \\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r \end{aligned}$$

where the expectations are conditioned on the initial state, S_0, and on the subsequent actions, $A_0, A_1, ..., A_{t-1}$, being taken according to π. The second and third equations hold if the steady-state distribution, $\mu_\pi(s) := \lim_{t\to\infty} Pr\{S_t = s | A_{0:t-1} \sim \pi\}$, exists and is independent of S_0, in other words, if the Markov decision problem (MDP) is ergodic. In an ergodic MDP, the starting state and any early decision made by the agent can have only a temporary effect; in the long run the expectation of being in a state depends only on the policy and the MDP transition probabilities. Ergodicity is sufficient but not necessary to guarantee the existence of the limit.

In the average-reward setting, returns are defined in terms of differences between rewards and the average reward:

$$G_t := R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots$$

This is known as the differential return, and the corresponding value functions are known as differential value functions. Differential value functions are defined in terms of the new return just as conventional value functions were defined in terms of the discounted return; thus we will use the same notation, $v_\pi(s) := \mathbb{E}_\pi [G_t | S_t = s]$ and $q_\pi(s,a) := \mathbb{E}_\pi [G_t | S_t = s, A_t = a]$ (similarly for v_* and q_*), for differential value functions. Differential value functions also have Bellman equations, just slightly different from those we have seen earlier. We simply remove all γs and replace all rewards by the difference between the reward and the true average reward:

$$\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \left[r - r(\pi) + v_\pi(s')\right] \\
q_\pi(s,a) &= \sum_{s',r} p(s',r|s,a) \left[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s',a')\right] \\
v_*(s) &= \max_a \sum_{s',r} p(s',r|s,a) \left[r - \max_\pi r(\pi) + v_*(s')\right] \\
q_*(s,a) &= \sum_{s',r} p(s',r|s,a) \left[r - \max_\pi r(\pi) + \max_{a'} q_*(s',a')\right]
\end{aligned}$$

There is also a differential form of the two TD errors:

$$\delta_t := R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)$$

and

$$\delta_t := R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)$$

where $\bar{R}_t$ is an estimate at time t of the average reward $r(\pi)$.

To approximate value function we'll write $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$. For example, $\hat{v}$ might be a linear function in features of the state, with $\boldsymbol{w}$ the vector of feature weights. More generally, $\hat{v}$ might be the function computed by a multi-layer artificial neural network, with $\boldsymbol{w}$ the vector of connection weights in all the layers.

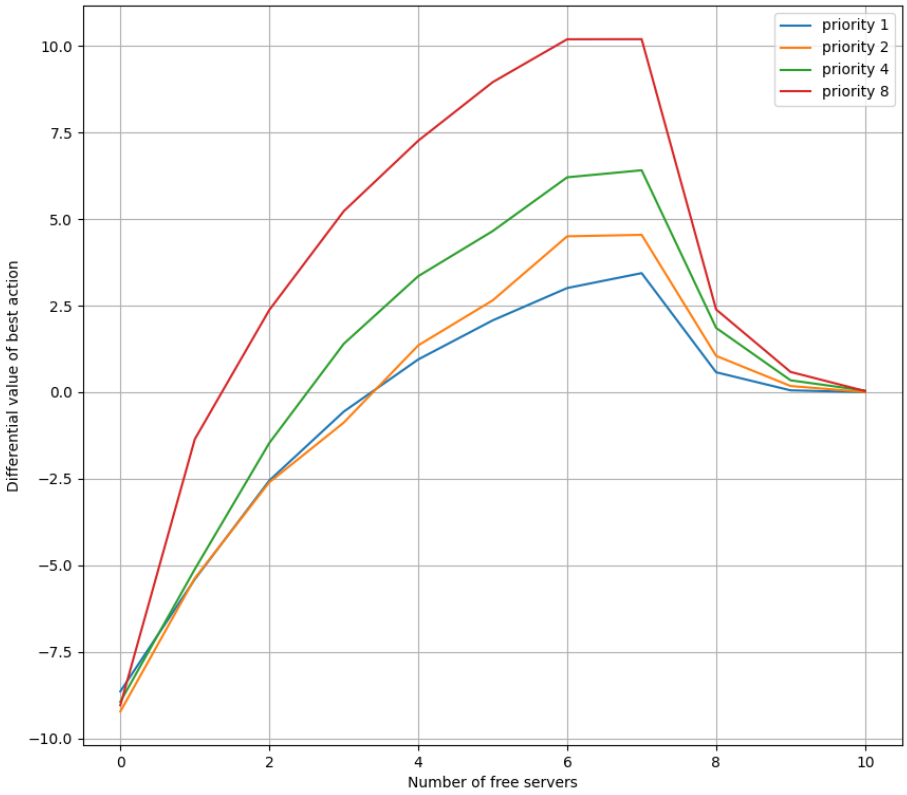
Typically, the number of weights (the dimensionality of $\boldsymbol{w}$) is much less than the number of states ($d \ll |S|$), and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such generalization makes the learning potentially more powerful but also potentially more difficult to manage and understand.

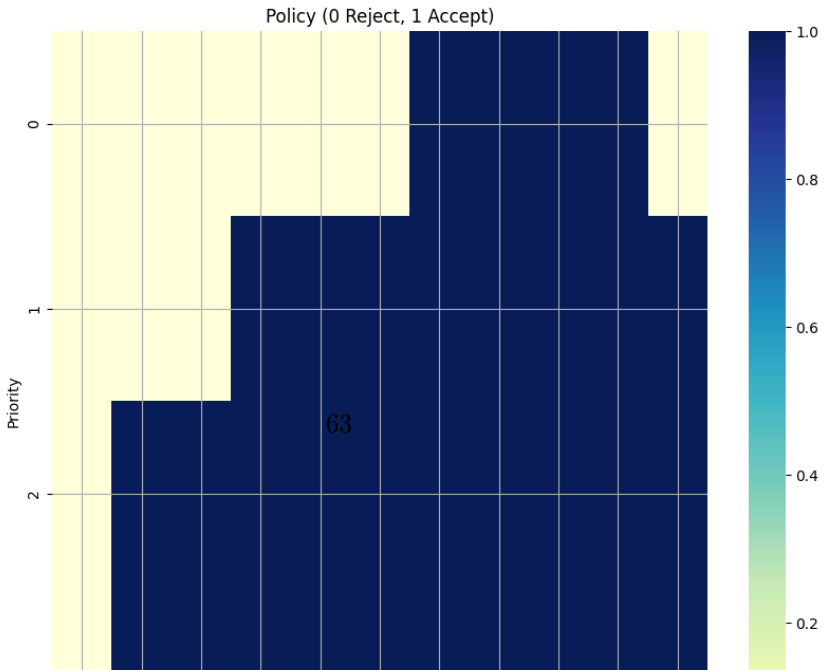
An average reward version of semi-gradient SARSA could be defined similar except with the differential version of the TD error. That is, by

$$w_{t+1} := w_t + \alpha\delta_t \nabla \hat{q}(S_t, A_t, w_t)$$

One limitation of this algorithm is that it does not converge to the differential values but to the differential values plus an arbitrary offset. Notice that the Bellman equations and TD errors given above are unaffected if all the values are shifted by the same amount. Thus, the offset may not matter in practice.

We have a differential action-value estimate for each pair of state (number of free servers and priority of the customer at the head of the queue) and action (accept or reject). The figure below shows the solution found by differential semi-gradient SARSA with parameters $\alpha = 0.01$, $\beta = 0.01$, and $\epsilon = 0.1$. The initial action values and $\bar{R}$ were 0.


priority 1
priority 2
priority 4
priority 8
Differential value of best action
Number of free servers
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
2
4
6
8
10


Policy (0 Reject, 1 Accept)
Priority
0
1
2
1.0
0.8
0.6
0.4
0.2

The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for $\bar{R}$ was about 2.37. (Note that priority 1 here is the lowest priority.)

▲

The $TD(\lambda)$ algorithm can be understood as one particular way of averaging n-step updates. This average contains all the n-step updates, each weighted proportionally to λ^{n-1} (where $\lambda \in [0,1]$), and is normalized by a factor of $1-\lambda$ to ensure that the weights sum to 1. The resulting update is toward a return, called the $\lambda - return$, defined in its state-based form by

$$G_t^\lambda := (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$$

If $\lambda = 0$, then the overall update reduces to its first component, the one-step TD update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the Monte Carlo update.

Semi-gradient $TD(\lambda)$ for estimating $\hat{v} \approx v_\pi$:

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(terminal, \cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize value-function weights $\boldsymbol{w}$ arbitrarily (e.g., $\boldsymbol{w} = 0$)
Loop for each episode:
  initialize S
  $\boldsymbol{z} \leftarrow 0$ (a d-dimensional vector)
  loop for each step of episode:
    choose $A \sim \pi(\cdot|S)$
    take action A, observe R, S'
    $\boldsymbol{z} \leftarrow \gamma\lambda\boldsymbol{z} + \nabla\hat{v}(S, \boldsymbol{w})$
    $\delta \leftarrow R + \gamma\hat{v}(S', \boldsymbol{w}) - \hat{v}(S, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\boldsymbol{z}$
    $S \leftarrow S'$
  until S' is terminal

This algorithm is not guaranteed to be stable when used with off-policy data and with a powerful function approximator. Several methods using eligibility traces have been proposed that achieve guarantees of stability under off-policy training. All the algorithms assume linear function approximation, though extensions to nonlinear function approximation can also be found in the literature. For example, $HTD(\lambda)$ is a hybrid state-value algorithm combining aspects of $GTD(\lambda)$ and $TD(\lambda)$. Its most appealing feature is that it is a strict generalization of $TD(\lambda)$ to off-policy learning, meaning that if the behavior policy happens to be the same as the target policy, then $HTD(\lambda)$ becomes the same as $TD(\lambda)$, which is not true for $GTD(\lambda)$. This is appealing because $TD(\lambda)$ is often faster than $GTD(\lambda)$ when both algorithms converge, and $TD(\lambda)$ requires setting only a single step size. $HTD(\lambda)$ is defined by

$$
\begin{aligned}
\boldsymbol{w}_{t+1} &:= \boldsymbol{w}_t + \alpha\delta_t^s \boldsymbol{z}_t + \alpha\left((\boldsymbol{z}_t - \boldsymbol{z}_t^b)^T \boldsymbol{v}_t\right)(\boldsymbol{x}_t - \gamma_{t+1}\boldsymbol{x}_{t+1}) \\
\boldsymbol{v}_{t+1} &:= \boldsymbol{v}_t + \beta\delta_t^s \boldsymbol{z}_t - \beta\left(\boldsymbol{z}_t^{bT} \boldsymbol{v}_t\right)(\boldsymbol{x}_t - \gamma_{t+1}\boldsymbol{x}_{t+1}), \quad with\, \boldsymbol{v}_0 := \mathbf{0} \\
\boldsymbol{z}_t &:= \rho_t(\gamma_t\lambda_t \boldsymbol{z}_{t-1} + \boldsymbol{x}_t), \quad \boldsymbol{z}_{-1} := \mathbf{0} \\
\boldsymbol{z}_t^b &:= \gamma_t\lambda_t \boldsymbol{z}_{t-1}^b + \boldsymbol{x}_t), \quad \boldsymbol{z}_{-1}^b := \mathbf{0}
\end{aligned}
$$

where $\beta > 0$ is a second step-size parameter. In addition to the second set of weights, $\boldsymbol{v}_t$, $HTD(\lambda)$ also has a second set of eligibility traces, $\boldsymbol{z}_t^b$. These are conventional accumulating eligibility traces for the behavior policy and become equal to $\boldsymbol{z}_t$ if all the ρ_t are 1, which causes the last term in the $\boldsymbol{w}_t$ update to be zero and the overall update to reduce to $TD(\lambda)$.

Methods that learn approximations to both policy and value functions are often called actor–critic methods, where 'actor' is a reference to the learned policy, and 'critic' refers to the learned value function, usually a state-value function. In other words, the steps of policy evaluation and policy improvement are not run to convergence, and instead the value function and policy are optimized jointly. One-step actor-critic methods:

$$
\begin{aligned}
\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\left(G_{t:t+1} - \hat{v}(S_t, \boldsymbol{w})\right)\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \\
&= \boldsymbol{\theta}_t + \alpha\left(R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})\right)\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \\
&= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}
\end{aligned}
$$

The generalizations to the forward view of n-step methods and then to a λ-return algorithm are straightforward.

Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_\theta \approx \pi_*$:

Input: a differentiable policy parametrization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parametrization $\hat{v}(s,\boldsymbol{w})$
Parameters: trace-decay rates $\lambda^\theta \in [0,1]$, $\lambda^w \in [0,1]$; step sizes $\alpha^\theta > 0$, $\alpha^w > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^d$ (e.g. to $\mathbf{0}$)
Loop forever (for each episode):
  initialize S (first state of episode)
  $\boldsymbol{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\boldsymbol{z}^{\boldsymbol{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot\,|S,\boldsymbol{\theta})$
    take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\boldsymbol{w}) - \hat{v}(S,\boldsymbol{w})$ (if S' is terminal then $\hat{v}(S',\boldsymbol{w}) \doteq 0$)
    $\boldsymbol{z}^{\boldsymbol{w}} \leftarrow \gamma\lambda^w \boldsymbol{z}^w + \nabla\hat{v}(S,\boldsymbol{w})$
    $\boldsymbol{z}^{\boldsymbol{\theta}} \leftarrow \gamma\lambda^\theta \boldsymbol{z}^\theta + I\nabla \ln\pi(A|S,\boldsymbol{\theta})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^w \delta \boldsymbol{z}^w$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^\theta \delta \boldsymbol{z}^\theta$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

13.3 Market Making

△

Let us consider a stochastic optimal control problem of a market maker trading in a weakly consistent, multivariate Hawkes process-based limit order book (LOB) mode. In the model the market responds dynamically to the market maker's control strategy - choice of placed limit and market orders by affecting the order arrival intensities.

A p-dimensional linear Hawkes process is a p-dimensional point process $N(t) = (N_k(t) : k = 1,\ldots,p)$ with the intensity of N_k (the k-th dimension) given by:

$$\lambda_k(t) = \mu_k + \sum_{l=1}^{p} \int_0^{t-} f_{k,l}(t-s) dN_l(s)$$

where $\mu_k \geq 0$ are the baseline intensities of underlying Poisson process, $N_l(t)$ the number of arrivals within $[0,t]$ corresponding to the l-th dimension, and $f_{k,l}(t)$ the kernels (triggering functions). Arrivals in dimension l perturb the intensity of the arrivals in dimension k at time t by $f_{k,l}(t-s)$ for $t > s$. We will use exponential kernels. Then the intensities are given by:

$$\lambda_k(t) = \mu_k + \sum_{l=1}^{p} \int_0^{t-} \alpha_{k,l} e^{-\beta_{k,l}(t-s)} dN_l(s)$$

where $\alpha_{k,l} \geq 0$ and $\beta_{k,l} \geq 0$ are the excitation and decay parameters.

Orders can be categorized as following.

type of order	mid-price	bid-price	ask-price
1 – aggressive market buy	+	no effect	+
2 – aggressive market sell	-	-	no effect
3 – aggressive limit buy	+	+	no effect
4 – aggressive limit sell	-	no effect	-
5 – aggressive limit buy cancellation	-	-	no effect
6 – aggressive limit sell cancellation	+	no effect	+
7 – non-aggressive market buy	no effect	no effect	no effect
8 – non-aggressive market sell	no effect	no effect	no effect
9 – non-aggressive limit buy	no effect	no effect	no effect
10 – non-aggressive limit sell	no effect	no effect	no effect
11 – non-aggressive limit buy cancellation	no effect	no effect	no effect
12 – non-aggressive limit sell cancellation	no effect	no effect	no effect

Untypical orders, such as iceberg orders or (partly) hidden orders, are omitted from consideration for simplicity's sake. Aggressive market orders, limit orders, and cancellations (events of type 1–6) affect either the bid or the ask price, and consequently the mid-price as well. Non-aggressive market orders (type 7–8) do not affect any of the prices; however, they generate trades and affect the volume at the top of the LOB and are hence relevant to MM. Non-aggressive limit orders and cancellations (type 9–12) affect neither the prices nor the volume at the top of the LOB and are hence ignored. Consequently, we consider 8 event types in total:

$$E_{all} = \{M_b^a, M_s^a, L_b^a, L_s^a, C_b^a, C_s^a, M_b^n, M_s^n\}$$

where the superscript denotes the (non)-aggressiveness, and the subscript the side (buy/sell). Allowing for variable jump sizes, the LOB dynamics are modeled by an 8-variate marked point process, where marks indicate jump sizes. The corresponding counting process is given by:

$$N(t) = \left(N_{M_b^a}(t), \ldots, N_{M_s^n}(t)\right)$$

and the associated intensity vector by:

$$\lambda(t) = \left(\lambda_{M_b^a}(t), \ldots, \lambda_{M_s^n}(t)\right)$$

After denoting the jump size (in ticks) associated with an individual event e by J_e, the mid-price P_t is given by:

$$P_t = P_0 + \left(\sum_{e, T(e) \in E_{inc}} J_e - \sum_{e, T(e) \in E_{dec}} J_e \right) \frac{\delta}{2}$$

where P_0 is the initial price, δ the tick size, $T(e)$ the type of event e, $E_{inc} = \{M_b^a, L_b^a, C_s^a\}$, and $E_{dec} = \{M_s^a, L_s^a, C_b^a\}$. Jump sizes are, for the sake of simplicity, assumed to be independent of the jump times and i.i.d.

The market making procedure is the following. At the start of each time-step, at time t, the agent (controller) cancels its outstanding limit orders (if there are any), and observes the state of the environment S_t containing market and agent-based features. The agent uses this information to select the action A_t - it decides whether to post limit orders (and at which prices) or a market order. If the absolute value of the agent's inventory is equal to the inventory constraint c, $c \in \mathbb{N}$, the order on the corresponding side is ignored. All relevant market (mid-price, bid and ask price, spread) and agent-based (inventory, cash) variables are then updated accordingly. Next, the LOB events generated by the simulation procedure are processed sequentially, which is followed by corresponding updates of the relevant variables. Executed limit orders cannot be replaced with new ones until the beginning of the next time-step. Finally, the agent reaches the end of the time-step and receives the reward $R_{t+\Delta t}$. When time $t + \Delta t$ is reached, unexecuted limit orders from the previous time-step are canceled, the agent observes the new state of the environment $S_{t+\Delta t}$, selects the action $A_{t+\Delta t}$ and the procedure is iterated until the terminal time T. The agent's inventory I_t is described by the following relation:

$$dI_t = dN_t^b - dN_t^a + dN_t^{mb} - dN_t^{ms}$$

where dN_t^b, dN_t^a, $dN_t^{mb}\, and\, dN_t^{ms}$ denote the number of the agent's limit bid (buy), limit ask (sell), market buy, and market sell orders executed up to time t, respectively. The process N_t^b is described by:

$$dN_t^b = dN_{M_s^a} \mathbb{1}_{fill, M_s^a} + dN_{M_s^n} \mathbb{1}_{fill, M_s^n}$$

where $\mathbb{1}_{fill, M_s^a}$ ($\mathbb{1}_{fill, M_s^n}$) is the indicator function for whether the incoming (non)-aggressive market order fills the market maker's limit order. The process N_t^a is described analogously. Finally, the agent's cash process X_t is given by:

$$dX_t = Q_t^a dN_t^a - Q_t^b dN_t^b - (P_t^a + \epsilon_t) dN_t^{mb} + (P_t^b - \epsilon_t) dN_t^{ms}$$

where Q_t^a (Q_t^b) denotes the price at which the agent's ask (bid) quote is posted, P_t^a (P_t^b) the best ask (bid) price, and ϵ_t the additional costs due to fees and market impact, all at time t. Furthermore, a number of simplifying assumptions are made:

- Market orders submitted by the market maker are aggressive with probability Z_1, and its limit order cancellations are aggressive with probability Z_2.
- Exponential distribution with density given by $f(x) = \frac{1}{\beta} \exp(-\frac{x-\mu}{\beta})$ is used for modeling the size of the price jumps J_e associated with aggressive events, where μ is the location and β the scale parameter. Where required (i.e. with aggressive limit orders and cancellations), the truncated exponential distribution with the corresponding parameters is used.

- Upon arrival of a non-aggressive market order, the probability of execution of the market maker's limit order standing at the best bid/ask price is fixed and given by Z_3. The limit order is either executed in its entirety or not at all.

The state space consists of the current inventory I_t, the current bid-ask spread Δt, and the trend variable α_t:

$$S_t = (I_t, \Delta_t, \alpha_t)$$

Due to the inventory constraints, there are $2c + 1$ possible inventory states, i.e., $I_t \in -c, \ldots, c$. The current bid-ask spread Δ_t is measured in ticks and strictly positive. The variable α_t accounts for the trend and is calculated as $\alpha_t = \lambda_{M_b^a}(t) + \lambda_{M_b^n}(t) - \lambda_{M_s^a}(t) - \lambda_{M_s^n}(t)$. The inventory feature I_t is normalized by the min-max normalization. Since features Δ_t and α_t have unknown means and variances, we generate a long trajectory with 100,000 steps by a controller that outputs completely random actions and use the obtained means and variances for z-score normalization.

The action (i.e. the controls) A_t at time t corresponds to a pair of offsets from the best bid (ask) prices. Hence:

$$A_t = (Q_t^a - P_t^a, P_t^b - Q_t^b)$$

The agent is allowed to post limit orders at all possible prices, thereby determining the level of aggressiveness/conservativeness on each of the sides of the LOB. If the agent posts a limit order pair with a negative quoted spread, the action is ignored. A bid (ask) order posted at or above (below) the best ask (bid) is treated as a buy (sell) market order and executed immediately. All limit and market orders sent by the market maker are assumed to be of unit size (i.e. are for one unit of the asset) and the controls are rounded to a multiple of the tick size.

The goal of the MM agent is to maximize the expectation of the terminal wealth while simultaneously minimizing the inventory risk. We assume that the market maker maximizes the following expression:

$$\mathbb{E}_\pi \left[W_T - \varphi \int_0^T |I_t| dt \right]$$

over the set of RL policies. Each policy $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$ maps states to probability distributions over the action space. $W_t = I_t P_t + X_t$ is the total wealth at time t, T the terminal time, and $\varphi \geq 0$ the running inventory penalty parameter which is used to disincentivize the market maker from holding non-zero positions and thereby exposing itself to the inventory risk. Note that we use the absolute value of the inventory instead of the quadratic inventory penalty to obtain a convenient value at risk (VaR) interpretation. Therefore, the reward at time $t + \Delta t$ is given by:

$$R_{t+\Delta t} = \Delta W_{t+\Delta t} - \varphi \int_{t}^{t+\Delta t} |I_s| ds$$

The integrand is piecewise constant and hence trivial.

A neural network with 2 fully-connected hidden layers of 64 neurons with ReLU activation is used to represent the controller. **Soft Actor-Critic** (SAC) is used to train the RL controller in an off-policy way in our experiments. It is a maximum entropy algorithm, meaning that it maximizes not only the return but also the policy entropy, thereby improving exploration. Then the optimal policy is

$$\pi^* = \arg\max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi} \left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot\,|s_t)\right)\right)\right] \tag{13.2}$$

where $\alpha > 0$ is the trade-off coefficient. V^π is now changed to include the entropy bonuses from every time step:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi} \left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot\,|s_t)\right)\right) \Big| s_0 = s \right] \tag{13.3}$$

Q^π is changed to include the entropy bonuses from every time step except the first:

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t H\left(\pi(\cdot\,|s_t)\right) \Big| s_0 = s, a_0 = a \right] \tag{13.4}$$

With these definitions, V^π and Q^π are connected by:

$$V^\pi(s) = \mathop{\mathbb{E}}_{a\sim\pi} \left[Q^\pi(s,a) \right] + \alpha H\left(\pi(\cdot\,|s_t)\right) \tag{13.5}$$

and the Bellman equation for Q^π is

$$\begin{aligned} Q^\pi(s,a) &= \mathop{\mathbb{E}}_{\substack{s'\sim P \\ a'\sim\pi}} \left[R(s,a,s') + \gamma \left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot\,|s')\right)\right)\right] \\ &= \mathop{\mathbb{E}}_{s'\sim P} \left[R(s,a,s') + \gamma V^\pi(s') \right] \end{aligned} \tag{13.6}$$

Rewriting this by using the definition of entropy (6.1) yields

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\substack{s'\sim P \\ a'\sim\pi}} \left[R(s,a,s') + \gamma \left(Q^\pi(s',a') - \alpha \log \pi(a'|s')\right)\right]$$

The right hand side is an expectation over next states (which come from the replay buffer) and next actions (which come from the current policy, and not the replay buffer). Since it's an expectation, we can approximate it with samples:

$$Q^{\pi}(s,a) \approx r + \gamma \left(Q^{\pi}(s', \tilde{a}') - \alpha \log \pi(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi(\cdot\,|s')$$

The next action notation is switched to $\tilde{a}'$, instead of a', to highlight that the next actions have to be sampled fresh from the policy (whereas by contrast, r and s' should come from the replay buffer).

SAC sets up the mean-squared Bellman error (MSBE) loss for each Q-function using this kind of sample approximation for the target. It also uses the clipped double-Q trick, and takes the minimum Q-value between the two Q approximators. Putting it all together, the loss functions for the Q-networks in SAC are:

$$L(\varphi_i, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_{\varphi_i}(s,a) - y(r,s',d)\right)^2\right]$$

where the target is given by

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{j=1,2} Q_{\varphi_{targ,j}}(s', \tilde{a}') - \alpha \log \pi_{\theta}(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_{\theta}(\cdot\,|s')$$

The policy should, in each state, act to maximize the expected future return plus expected future entropy (13.5). The way we optimize the policy makes use of the reparameterization trick, in which a sample from $\pi_{\theta}(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise. To illustrate: following the authors of the SAC paper, we use a squashed Gaussian policy, which means that samples are obtained according to

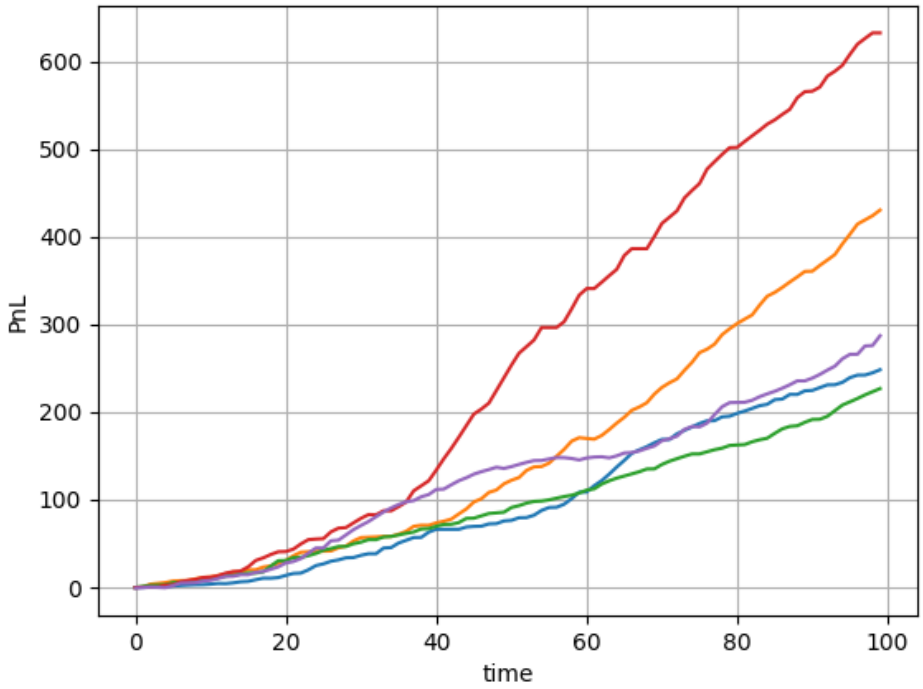
$$\tilde{a}_{\theta}(s,\xi) = \tanh\left(\mu_{\theta}(s) + \sigma_{\theta}(s) \odot \xi\right), \quad \xi \sim \mathcal{N}(0, I)$$

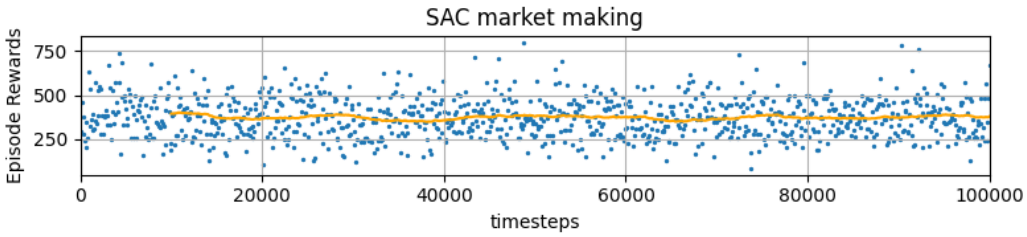
The reparameterization trick allows us to rewrite the expectation over actions (which contains a pain point: the distribution depends on the policy parameters) into an expectation over noise (which removes the pain point: the distribution now has no dependence on parameters):

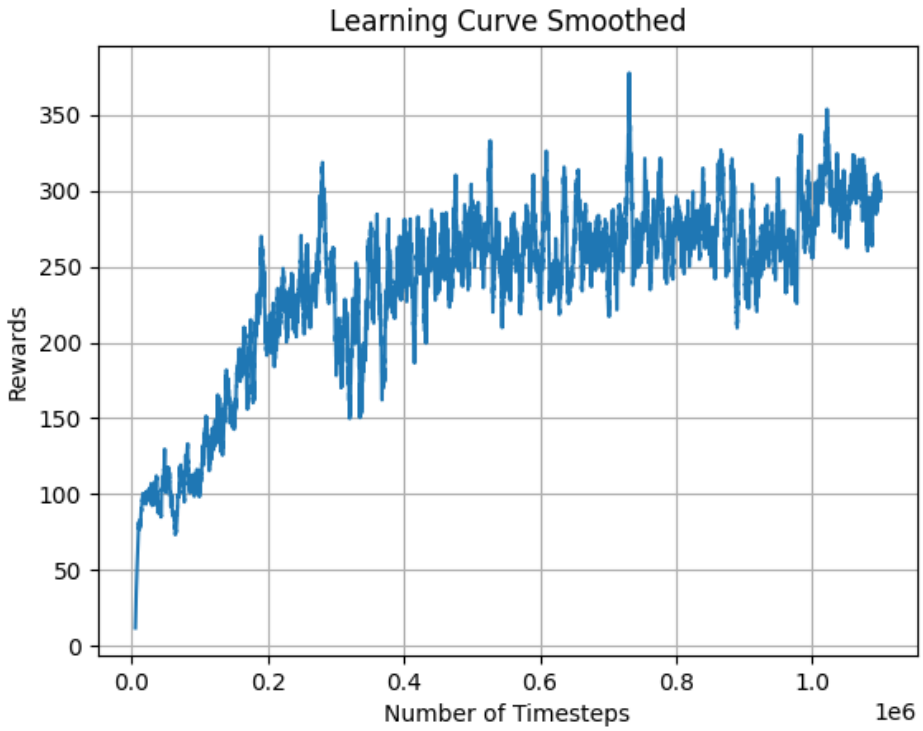
$$\underset{a\sim\pi}{\mathbb{E}}\left[Q^{\pi_{\theta}}(s,a) - \alpha \log \pi_{\theta}(a|s)\right] = \underset{\xi\sim\mathcal{N}}{\mathbb{E}}\left[Q^{\pi_{\theta}}(s, \tilde{a}_{\theta}(s,\xi)) - \alpha \log \pi_{\theta}(\tilde{a}_{\theta}(s,\xi)|s)\right]$$

To get the policy loss, the final step is that we need to substitute $Q^{\pi_{\theta}}$ with one of our function approximators which is the minimum of the two Q approximators. The policy is thus optimized according to

$$\max_{\theta} \underset{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}}}{\mathbb{E}}\left[\min_{j=1,2} Q_{\varphi_j}\left(s, \tilde{a}_{\theta}(s,\xi)\right) - \alpha \log \pi_{\theta}(\tilde{a}_{\theta}(s,\xi)|s)\right]$$


PnL
time
0
100
200
300
400
500
600
0
20
40
60
80
100


SAC market making
Episode Rewards
timesteps
250
500
750
0
20000
40000
60000
80000
100000



▲

14 Miscellaneous

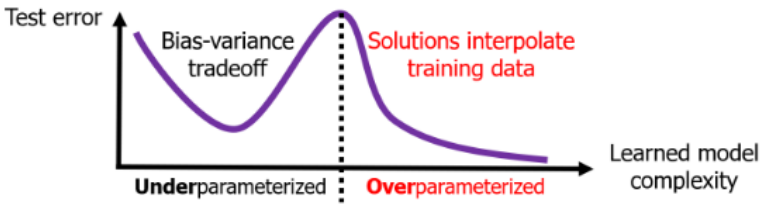


Figure 1: Double descent of test errors (i.e., generalization errors) with respect to the complexity of the learned model. TOPML studies often consider settings in which the learned model complexity is expressed as the number of (independently tunable) parameters in the model. In this qualitative demonstration, the global minimum of the test error is achieved by maximal overparameterization.

References

[1] S. Bird, E. Klein, E. Loper "Natural Language Processing with Python", 2009

[2] C. Manning, H. Schütze "Foundations of Statistical Natural Language Processing", 1999

[3] T.Bunk, D.Varshneya, V.Vlasov, A.Nichol "DIET: Lightweight Language Understanding for Dialogue Systems", 2020

[4] D.Jurafsky, J.Martin "Speech and Language Processing", 2020

[5] T.Mikolov, K.Chen, G.Corrado, J.Dean "Efficient Estimation of Word Representations in Vector Space", 2013

[6] T.Mikolov, I.Sutskever, K.Chen, G.Corrado, J.Dean "Distributed Representations of Words and Phrases and their Compositionality", 2013

[7] Y.Goldberg, O.Levy "word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method", 2014

[8] D.Blei, A.Ng, M.Jordan "Latent Dirichlet allocation", 2003

[9] J.Pennington, R.Socher, C.Manning "Glove: Global vectors for word representation", 2014

[10] J.Devlin, M.Chang, K.Lee, K.Toutanova "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

[11] A.Vaswani, N.Shazeer, N.Parmar, J.Uszkoreit, L.Jones, A.Gomez, L.Kaiser, I.Polosukhin "Attention Is All You Need", 2017

[12] L.Wu, A.Fisch, S.Chopra, K.Adams, A.Bordes, J.Weston "StarSpace: Embed All The Things!", 2017

[13] "The Annotated Transformer", 2018

[14] V.Vlasov, J.Mosig, A.Nichol "Dialogue Transformers", 2020

[15] C.Bishop "Pattern recognition and machine learning", 2006

[16] I.Goodfellow, Y.Bengio, A.Courville "Deep Learning", 2016

[17] Y.Dar, V.Muthukumar, R.Baraniuk "A Farewell to the Bias-Variance Tradeoff? An Overview of the Theory of Overparameterized Machine Learning", 2021

[18] A.Trask "Grokking Deep Learning", 2019

[19] M.Jaderberg, K.Simonyan, A.Zisserman, K.Kavukcuoglu "Spatial Transformer Networks", 2016

[20] S.Ioffe, C.Szegedy "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", 2015

[21] F.Schroff, D.Kalenichenko, J.Philbin "FaceNet: A Unified Embedding for Face Recognition and Clustering", 2015

[22] F.Chollet "Xception: Deep Learning with Depthwise Separable Convolutions", 2017

[23] S.Russell, P.Norvig "Artificial Intelligence. A Modern Approach", 4 ed., 2021

[24] R.Sutton, A.Barto "Reinforcement Learning: An Introduction", 2 ed., 2020

[25] B.Gašperov, Z.Kostanjčar "Deep Reinforcement Learning for Market Making Under a Hawkes Process-Based Limit Order Book Model", 2022

[26] T.Haarnoja, A.Zhou, P.Abbeel, S.Levine "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor", 2018

[27] Soft Actor-Critic; https://spinningup.openai.com/en/latest/algorithms/sac.html

[28] D.Kingma, M.Welling "Auto-Encoding Variational Bayes", 2013