MIT 6.86x

Gunbir Singh Baveja

May 16, 2023

1 Linear Classifiers and Generalizations

1.1 Intro

Training data can be graphically depicted on a (hyper)plane. **Classifiers** are mappings taht take feature vectors as input and produce labels as output. A common kind of classifier is the lilenar classifier, which linearly divides space (the (hyper)plane where training data lies) into two. Given a point x in the space, the classifier h outputs $h(x) = 1$ or $h(x) = -1$, depending on where the point x exists in among the two linearly divided spaces.

What is the meaning of hypothesis space? It is the set of possible classifiers.

1.2 Linear Classifier and Perceptron

- Feature, vectors - $x \in \mathbb{R}^d, y \in \{-1, 1\}$
- Training set - $S_n = \{\left(x^{(i)}, y^{(i)}\right), i = 1, \ldots, n\}$
- Classifier - $h : \mathbb{R}^d \to \{-1, 1\}, h(x) = 1, \mathcal{X}^+ = \{x \in \mathbb{R}^d : h(x) = 1\}, \mathcal{X}^- = \{x \in \mathbb{R}^d : h(x) = -1\}$
- Training error - $\mathcal{E}_n(h) = \frac{1}{n} \sum_{i=1}^{n} [\![h\left(x^{(i)}\right) \neq y^{(i)}]\!] = 1$ if error and 0 otherwise.
- Test error - $\mathcal{E}(h)$
- Set of classifiers - $\mathcal{H}, h \in \mathcal{H}$
- **NEW** Perceptron Algorithm - $\hat{h} = \mathcal{A}\left(S_n, \mathcal{H}\right)$

During most of the lectures, for a linear classifier h, $h(x; \theta) = sign(\theta \cdot x), \theta \in \mathbb{R}^d$, i.e., the sign of the dot product of θ and x. The decision boundary can be represented as $\{x : \theta_1 x_1 + \theta_2 x_2 = 0\}$ which can be represented in terms of vectors $\theta \cdot X = 0$ where $\theta = [\theta_1, \theta_2]^T, X = [x_1, x_2]^T$. **Note:** The decision boundary talked about above does not have θ_0 as we're only talking about linear classifiers that pass through the origin at the moment.

If we introduce the offset parameter θ_0, the new vector now becomes $\theta = [\theta_1, \theta_2, \cdots, \theta_0]^T$ and the classifier/decision boundary equation now becomes $\{x : \theta \cdot \mathbf{x} + \theta_0\}$ and the new linear classifier now becomes $h(x; \theta, \theta_0) = sign(\theta \cdot \mathbf{x} + \theta_0), \theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$.

Definition: Training examples $S_n = \{\left(x^{(i)}, y^{(i)}\right), i = 1, \ldots, n\}$ are *linearly separable* if there exists a parameter vector $\hat{\theta}$ and offset parameter $\hat{\theta_0}$ such that $y^{(i)}\left(\hat{\theta} \cdot x^{(i)} + \hat{\theta_0}\right) > 0$ for all $i = 1, \ldots, n$.

Algorithm 1 Perceptron Algorithm

procedure PERCEPTRON($\left(\{x^{(i)}, i = 1, \ldots, n\}, T\right)$)
  Initialize $\theta = 0$ (vector)
  for $t = 1, \ldots, T$ **do**
    for $i = 1, \ldots, n$ **do**
      if $y^{(i)}\left(\theta \cdot x^{(i)}\right) \leq 0$ **then**
        update $\theta = \theta + y^{(i)} x^{(i)}$

1.3 Hinge Loss, Margin boundaries

Consider a line L in $\mathbb{R}^2$ given by the equation

$$L : \theta \cdot x + \theta_0 = 0$$

where θ is a vector normal to the line L, then the shortest distance d between the line L and a point P is

$$d = \frac{|\theta \cdot x_0 + \theta_0|}{\|\theta\|}$$

The **margin boundary** is the set of points x which satisfy

$$\theta \cdot x + \theta_0 = \pm 1.$$

So, the distance from the decision boundary to the margin boundary $\frac{1}{\|\theta\|}$. As we increase $\|\theta\|$, the margin boundaries move closer to the decision boundary.

- Hinge Loss - $\text{Loss}_h\left(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)\right) = \begin{cases} 0 & \text{if } z \geq 1 \\ 1 - z & \text{if } z < 1 \end{cases}$
- Regularization: towards max margin

$$\max \frac{1}{\|\theta\|} \qquad \min \frac{1}{2}\|\theta\|^2$$

- The objective

$$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}_h\left(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)\right) + \frac{\lambda}{2}\|\theta\|^2.$$

- General optimization formulation of learning

$$\text{objective function} = \text{average loss} + \text{regualarization}$$

A little insight to Generalization Generalization can be understood by dividing the objective function by lambda and remove its dependence on the margin-maximizing term.

$$\frac{1}{\lambda} J(\theta, \theta_0) = \frac{1}{\lambda n} \sum_{i=1}^{n} \text{Loss}_h \left(y^{(i)} (\theta \cdot x^{(i)} + \theta_0) \right) + \frac{\|\theta\|^2}{2}.$$

Here, $1/n\lambda$ can be written as C. Now, plotting the objective function loss against C, there is an infinimum that can be observed. Towards the left of the infinimum, we have underfitting where the training loss is getting lowered and so is the testing loss, and toward the right we have overfitting, where the training and test loss both are getting higher.

The infinimum can be denoted as C^* which minimizes the sum of training and testing loss.

1.3.1 Error Decomposition

For training examples: $\mathcal{S} : (\mathcal{X} \times \mathcal{Y})^N$ from a dataset distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$. A hypothesis $f : \mathcal{X} \to \mathcal{Y}$ has **training error** $\text{trn}_{\mathcal{S}}(f) = \mathbb{P}_{(x,y)\sim\mathcal{S}}[f(x) \neq y]$, an average over samples; and **testing error** $\text{tst}(f) = \mathbb{P}_{(x,y)\sim\mathcal{D}}[f(x) \neq y]$, an average over the dataset.

A *learning program* is a function $\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \mathcal{Y})$; we want to design $\mathcal{L}$ so that it maps typical $\mathcal{S}$'s to fs with low $\text{tst}(f)$.

So we often define $\mathcal{L}$ to roughly minimize $\text{trn}_{\mathcal{S}}$ over a set $\mathcal{H} \subset (\mathcal{X} \to \mathcal{Y})$ of the candidate patterns. Then tst decomposes into the failures of $\text{trn}_{\mathcal{S}}$ to estimate tst (generalization), of $\mathcal{L}$ to minize $\text{trn}_{\mathcal{S}}$ (optimization), and of $\mathcal{H}$ to contain the original dataset (approximation/representation):

$$\begin{aligned}
\text{tst}(\mathcal{L}(\mathcal{S})) = \text{tst}(\mathcal{L}(\mathcal{S})) \quad & -\text{trn}_{\mathcal{S}}(\mathcal{L}(\mathcal{S})) && \text{(generalization error)} \\
+ \text{trn}_{\mathcal{S}}(\mathcal{L}(\mathcal{S})) \quad & -\text{inf}_{\mathcal{H}}(\text{trn}_{\mathcal{S}}(f)) && \text{(optimization error)} \\
+ \text{inf}_{\mathcal{H}}(\text{trn}_{\mathcal{S}}(f)) & && \text{(approximation error)}
\end{aligned}$$

These terms are in tension. For example, as $\mathcal{H}$ grows, the approx. error may decrease while the gen. error may increase – this is the "**bias-variance** tradeoff".

1.3.2 Gradient Descent: Geometrically Revisited

Assume $\theta \in \mathbb{R}$. Our goal is to find θ that minimizes

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h \left(y^{(i)} (\theta \cdot x^{(i)} + \theta_0) \right) + \frac{\lambda}{2} \|\theta\|^2$$

through gradient descent.

In other words, we will

1. Start θ at an arbitrary location: $\theta \leftarrow \theta_{start}$
2. Update θ repeatedly with $\theta \leftarrow \theta - \eta \frac{\partial J(\theta, \theta_0)}{\partial \theta}$ until θ does not change significantly.

Stochastic Gradient descent With stochastic gradient descent, we choose $i \in \{1, \ldots, n\}$ at random and update θ such that

$$\theta \leftarrow \theta - \eta \nabla_\theta \left[\text{Loss}_h \left(y(i) \cdot x^{(i)} + \theta_0 \right) \right) + \frac{\lambda}{2} \|\theta\|^2 \Bigg]^2 .$$

Note that when talking about loss, we refer to the *Hinge loss* which can be written formally as

$$\max(0, 1 - y \cdot f(x))$$

Note In a realizable case, if we only disallow any margin violations, the quadratic program we have to solve is to find θ, θ_0 that minimizes $\frac{1}{2}\|\theta\|^2$ subject to the $y(\theta \cdot \mathbf{x} + \theta_0) \geq 1$.

1.4 A look at Cross Validation

If we have a dataset with training and validation data points, $(\mathcal{X}_{\text{data}}, \mathcal{Y}_{\text{data}})$, we take a parameter n which accounts for the n total divisions we make of the total dataset such that each small subset is still a good representaion of its whole.

We take an array of α and iterate through the partitions classifying training with validation dataset to get an accuracy $S(\alpha)$. We repeat this n times and get $S(\alpha_1) = \frac{1}{n} \sum_{n=1}^{n} S_n(\alpha_1)$.

We repeat the above process for each $\alpha_i \in \alpha$ which would give us $\mathcal{S}(a)$ and the goal becomes to find the $\arg\max \mathcal{S}(\alpha)$ and find α^*.

2 Nonlinear Classifications and Linear Regression

2.1 Linear Regression

Instead of having a binary output $y \in \{0, 1\}$, linear regression can be defined as a function:

$$f(x, \theta, \theta_0) = \sum_{i=1}^{d} \theta_i x_i + \theta_0 = \theta \mathbf{x} + \theta_0,$$

where we take $\theta_0 = 0$ for simplification, and $y \in \mathbb{R}$.

Note We see that since $f(x)$ is a linear function, it would only be useful for predicting linearly shaped data and that is true. We will see in section XX that we can map the data into higher dimensional representations and then linearly solve them.

2.1.1 Objective

The empirical risk R_n is defined as

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}\left(y^{(t)} - \theta \cdot x^{(t)}\right)$$

where $\left(x^{(t)}, y^{(t)}\right)$ is the tth training example (with n in total), and Loss is some loss function, e.g. the hinge loss or the squared loss given by $(z) = z^2/2$.

Recall that the hinge loss is defined as

$$\text{Loss}_h(z) = \begin{cases} 0 & \text{if } z \geq 1 \\ 1 - z, & \text{otherwise} \end{cases}.$$

Mistakes

1. **Structural Mistakes:** If the true underlying relationship is of a different *structure* than the regression function, there is a high structural loss that is incurred. For example, if the data is non-linear but the regression function is linear.

2. **Estimation Mistakes:** This is the same as the training loss, and is usually incurred when the training dataset is very small. Increasing the number of training examples decreases this error.

NOTE!! There is a trade-off between these two mistakes. Using a rich high-dimensional function for regression might reduce structural mistakes, but the same dataset would be used to train many more parameters thereby incurring an estimation loss. On the other hand, if you have the minimum number of parameters to help estimation error, structural error is introduced.

2.1.2 Gradient-based approach

$$\nabla_\theta \left(y^{(t)} - \theta x^{(t)}\right)^2 /2 = \left(y^{(t)} - \theta x^{(t)}\right) \nabla_\theta \left(y^{(t)} - \theta x^{(t)}\right) = -\left(y^{(t)} - \theta x^{(t)}\right) x^{(t)}$$

Algorithm 2 Gradient Descent

Initialize $\theta = 0$
Randomly pick $t = \{1, \ldots, n\}$
$\theta = \theta + \eta \left(y^{(t)} - \theta x^{(t)}\right) \cdot x^{(t)}$
$\eta_t = \frac{1}{1+t}$

2.1.3 Closed form solution

There are two caveats to this:

1. Is A always reversible? We see that A is only reversible if the vectors of A, $x^1, x^2, \ldots, x^n$ span $\mathbb{R}^d$, and $n >> d$. Therefore, it becomes unlikely when the number of training examples is much larger than the dimensionality of the feature vector.

2. Reversing the matrix is of complexity $\mathcal{O}(d^3)$ and so as the dimensionality vector increases, the cost increases much more.

The empirical risk is

$$R_n(\theta) = \frac{1}{n} \sum_{t=1}^{n} \left(y^{(t)} - \theta x^{(t)}\right)^2 /2.$$

$$\begin{aligned}
\nabla_\theta R_n(\theta)\big|_{\theta=\vec{\theta}} &= \frac{1}{n} \sum_{t=1}^{n} \left(y^{(t)} - \theta x^{(t)}\right)^2 /2\big|_{\theta=\vec{\theta}} \\
&= -\frac{1}{n} \sum_{t=1}^{n} \left(y^{(t)} - \vec{\theta} x^{(t)}\right) x^{(t)} \\
&= \underbrace{-\frac{1}{n} \sum_{t=1}^{n} y^{(t)} x^{(t)}}_{\text{b}} + \frac{1}{n} \sum_{t=1}^{n} x^{(t)} x^{(t)} \\
&= -b + \frac{1}{n} \sum_{t=1}^{n} x^{(t)} \vec{\theta} x^{(t)} \\
&= -b + \underbrace{\frac{1}{n} \sum_{t=1}^{n} x^{(t)} \left(x^{(t)}\right)^T \vec{\theta}}_{\text{A}} \\
&\implies -b + A\vec{\theta} = 0 \\
&\implies A\vec{\theta} = b \\
&\implies \vec{\theta} = A^{-1} b.
\end{aligned}$$

2.1.4 Regularization

Regularization in regression tries to push the parameters closer to 0 and it only happens if the move is justified (training loss isn't incurred too much).

We define a new objective function for regression known as **Ridge Regression** as

$$J_{n,\lambda}(\theta, \theta_0) = \frac{1}{n} \sum_{t=1}^{n} \frac{\left(y^{(t)} - \theta \cdot x^{(t)} - \theta_0\right)^2}{2} + \frac{\lambda}{2}\|\theta\|^2.$$

We see that the gradient of this loss function is

$$\nabla_\theta J_{n,\lambda}(\theta, \theta_0) = \nabla_\theta \left(\frac{\lambda}{2}\|\theta\|^2 + \left(y^{(t)} - \theta x^{(t)}\right)^2 /2 \right) = \lambda\theta - \left(y^{(t)} - \theta x^{(t)}\right) x^{(t)}$$

Algorithm 3 Ridge Gradient Descent

Initialize $\theta = 0$
Randomly pick $t = \{1, \ldots, n\}$
$\theta = \theta - \eta \left(\lambda\theta - \left(y^{(t)} - \theta x^{(t)}\right) x^{(t)}\right) = \theta(1 - \eta\lambda) + \eta \left(y^{(t)} - \theta x^{(t)}\right) x^{(t)}$
$\eta_t = \frac{1}{1+t}$

2.1.5 Equivalance of regularization to a Gaussian Prior on Weights

The regularized linear regression can be interpreted from a probabilistic point of view. Suppose we are fitting a linear regression model with n data points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Let's assume the ground truth is that y is linearly related to x but we also observed some noise ϵ for y:

$$y_t = \theta \cdot x_t + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

Then the likelihood of our observed data (assuming that they are independent) is

$$\prod_{t=1}^{n} \mathcal{N}(y_t \mid \theta x_t, \sigma^2).$$

Now, if we impose a Gaussian prior $\mathcal{N}(\theta \mid 0, \lambda^{-1})$, the likelihood will change to

$$\prod_{t=1}^{n} \mathcal{N}(y_t \mid \theta x_t, \sigma^2)\mathcal{N}(\theta \mid 0, \lambda^{-1}).$$

Take the logarithm of the likelihood, we will end up with

$$\sum_{t=1}^{n} -\frac{1}{2\sigma^2}(y_t - \theta x_t)^2 - \frac{1}{2}\lambda\|\theta\|^2 + \text{ constant}.$$

2.2 Nonlinear Classification

We can use linear classifiers by mapping all examples $x \in \mathbb{R}^d$ to different feature vectors $\phi(x) \in \mathbb{R}^p$ where typically p is much larger than d. We would then simply use a linear classifier on the new (higher dimensional) feature vectors, pretending that they were the original input vectors. As a result, all the linear classifiers we have learned remain applicable, yet produce non-linear classifiers in the original coordinates.

There are many ways to create such feature vectors. One common way is to use polynomial terms of the original coordinates as the components of the feature vectors. We have seen two examples in the video above. We will recall the 1-dimensional example here and see another 2-dimensional example in the problem below.

Remember that $h(x; \theta, \theta_0) = \text{sign}(\theta x + \theta_0)$. We map x such that

$$\begin{aligned} x \in \mathbb{R} \to \phi(x) \in \mathbb{R}^2 &= \begin{bmatrix} x \\ x^2 \end{bmatrix}, \\ \theta \to \theta &= \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}, \\ h(x; \theta, \theta_0) \to h &= \text{sign}(\theta \cdot \phi(x) + \theta_0) \\ &= \text{sign}(\theta_1 x + \theta_2 x^2 + \theta_0) \end{aligned}$$

The new feature vector $\phi(x)$ thus equals

$$\begin{bmatrix} x(\phi_1) \\ x^2(\phi_2) \end{bmatrix}.$$

We can add more polynomial terms

$$x \in \mathbb{R}, \phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \\ \vdots \end{bmatrix}.$$

This means lots of features in higher dimensions

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2, \phi(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ \sqrt{2} x_1 x_2 \end{bmatrix} \in \mathbb{R}^5.$$

Why not feature vectors?

- By mapping input examples explicitly into feature vectors, and performing linear calssification or regression on top of such feature vectors, we get a lot of expressive power.
- By the downside is that these vectors can be quite high dimensional.

2.2.1 Kernels

Computing the inner product between two feature vectors *can be* cheap even if the vectors are very high dimensional

$$\phi(x) = [x_1, x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2]^T \qquad \phi(x') = [x'_1, x'_2, {x'_1}^2, \sqrt{2}x'_1x'_2, {x'_2}^2]^T$$

Taking a dot product leads to $\phi(x) \cdot \phi(x') = (x \cdot x') + (x \cdot x')^2$ which is much easier to compute! This is the kernel function.

For some feature maps, we can evaluate the inner products very efficiently, e.g.,

$$K(x, x') = \phi(x)\phi(x') = (1 + x \cdot x')^P, \text{where } p = 1, 2, \ldots.$$

In those cases, it is advantageous to express the linear classifiers (regression methods) in terms of kernels rather than explicitly constructing feature vectors.

Algorithm 4 Recall Perceptron (The kernel Perceptron Algorithm)

procedure KERNEL PERCEPTRON($\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n, T\}$)
  Initialize $\alpha_1, \alpha_2, \ldots, \alpha_n$ to some values;
  for $t = 1, \ldots, T$ **do**
    for $i = 1, \ldots, n$ **do**
      if (Mistake Condition Expressed in α_j) **then**
        Update α_j accordingly

We can summarize $\theta = \sum_{j=1}^{n} \underbrace{\alpha_j}_{\#\text{ of mistakes}} y^{(i)}\phi(x^{(i)})$

Composition Rules

1. $K(x, x') = 1$ is a kernel function.
2. Pre or post-multiplying a function $f : \mathbb{R}^d \to \mathbb{R}$ and $K(x, x')$ is a kernel, then $\tilde{K}(x, x') = f(x)K(x, x')f(x')$.
3. If $K_1(x, x')$ and $K_2(x, x')$ are kernels, then $K(x, x') = K_1(x, x')+K_2(x, x')$ is a kernel.
4. If $K_1(x, x')$ and $K_2(x, x')$ are kernels, then $K(x, x') = K_1(x, x')K_2(x, x')$ is a kernel.

2.2.2 Radial Basis Kernel

The feature vectors can be infinite dimensional, this emans that they have unlimited expressive power. One such kernel that allows for compact representation of examples is the **radial basis kernel**:

$$K(x, x') = \exp\left(-\frac{1}{2\sigma^2}\|x - x'\|^2\right),$$

where $\|x - x'\|^2$ may be recognized as the squared Euclidean distance between the two feature vectors, and σ a free parameter. An equivalent definition involves a paramter $\gamma = \frac{1}{2\sigma^2}$:

$$K(x, x') = \exp(-\gamma\|x - x'\|^2).$$

The RBF value decreases between 0 and 1 (where $x = x'$). The feature space of the kernel has an infinite number of dimensions[1]; for $\sigma = 1$, its expansion using the multinomial theorem is:

$$\begin{aligned}
\exp\left(-\frac{1}{2}\|x - x'\|^2\right) &= \exp\left(\frac{2}{2}x^T x' - \frac{1}{2}\|x\|^2 - \frac{1}{2}\|x'\|^2\right) \\
&= \exp(x^T x')\exp\left(-\frac{1}{2}\|x\|^2\right)\exp\left(-\frac{1}{2}\|x'\|^2\right) \\
&= \sum_{j=1}^{\infty}\frac{(x^T x')^j}{j!}\exp\left(-\frac{1}{2}\|x\|^2\right)\exp\left(-\frac{1}{2}\|x'\|^2\right) \\
&= \sum_{j=0}^{\infty}\sum_{n_1+n_2+\cdots+n_k=j}\exp\left(-\frac{1}{2}\|x\|^2\right)\frac{x_1^{n_1}\cdots_k^{n_k}}{\sqrt{n_1!\cdots n_k!}}\exp\left(-\frac{1}{2}\|x'\|^2\right)\frac{{x'_1}^{n_1}\cdots{x'_k}^{n_k}}{\sqrt{n_1!\cdot n_k!}} \\
&= \langle\varphi(x), \varphi(x')\rangle \\
\varphi(x) &= \exp\left(-\frac{1}{2}\|x\|^2\right)\left(a_{l_0}^{(0)}, a_1^{(1)}, \ldots, a_{l_1}^{(1)}, \ldots, a_1^{(j)}, \ldots, a_{l_j}^{(j)}, \ldots\right)
\end{aligned}$$

where $l_j = \binom{k+j-1}{j}, a_l^{(j)} = \frac{x_1^{n_1}\cdots x_k^{n_k}}{\sqrt{n_1!\cdots n_k!}}$ and $n_1 + n_2 + \cdots n_k = j \wedge 1 \leq l \leq l_j$.

3 Neural Networks

3.1 Feedforward Neural Nets

- Units in neural networks are linear classifiers, just with different output non-linearity
- The units in feedforward neural networks are arranged in layers
- By learning the parameters associated with the hidden layer units, we learn how to *represent* examples (as hidden layer activations)
- The representations in neural networks are learned directly to facilitate the end-to-end task
- A simple classifier (output unit) suffices to solve complex classification tasks if it operates on the hidden layer representations

[1]For further reading: check this out

Learning neural networks With $f = f(x; w)$ being the output layer functions creating the output y, we update the weights by:

$$w_{ij}^{'} \leftarrow w_{ij}^{'} - \eta \frac{\partial \text{Loss}\,(y, f(x; w))}{\partial w_{ij}^{'}}.$$

And for calculating the derivative, we use **back-propagation**.

3.1.1 Back-propagation

In order to calculate the effectiveness of a weight w that changes f_{L-1} to f_L, we say that

$$\frac{\partial f_L}{\partial f_{L-1}}$$

communicates the information needed to train the weights.

Using similar intuition, we say that the gradient is calculated via:

$$\frac{\partial \text{Loss}}{\partial w_i} = \frac{\partial f_i}{\partial w_i} \times \frac{\partial \text{Loss}}{\partial f_i}.$$

Here,

$$\frac{\partial \text{Loss}}{\partial f_i} = \frac{\partial f_{i+1}}{\partial f_i} \times \frac{\partial \text{Loss}}{\partial f_{i+1}}.$$

Proof for the derivative of the hyberbolic tangent function

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}z}\left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right) &= \frac{(e^z - e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\
&= \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2} \\
&= 1 - \left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)^2.
\end{aligned}$$

Using the derivative of the hyperbolic tangent functions, we can find the derivative $\frac{\partial f_{i+1}}{\partial f_i} = (1 - f_{i+1}^2) w_{i+1}$ and $\frac{\partial f_i}{\partial w_i} = (1 - f_i^2) x$.

Thus, back-propagation becomes a recursive function where the base case is $\frac{\partial \text{Loss}}{\partial f_L} = \frac{\partial \frac{1}{2}(y - f_L)^2}{\partial f_L}$ which equals $-(y - f_L)$, where f_L is the function of the last neuron in a network.

Note If the derivatives of this *Jacobian vector* becomes too small, then the gradient *vanishes* pretty quickly. Conversely, if the derivatives are too large, the gradients can *explode*. This is known as the **vanishing and exploding gradient problem**.

Side Note In these lectures, we take the loss function to be $\mathcal{L}(y, f_L) = (y - f_L)^2$.

- Neural networks can be learned with SGD similarly to linear classifiers
- The derivatives necessary for SGD can be evaluated effectively via back-propagation
- Multi-layer neural network models are complicated... we are no longer guaranteed to reach gloabl (only local) optimum with SGD
- Large models tend to be easier to learn... units only need to be adjusted so that they are, collectively, sufficient to solve the task.

3.2 Recurrent Neural Networks

There are two key concepts which can be understood using the following MT example:

$$\text{I like oranges very much} : \underbrace{[0, -2, 0.3, 4, 0, 0, 1, 2]}_{\text{encoding}} \rightarrow \underbrace{\text{Me gustan las naranjas mucho}}_{\text{decoding}}.$$

3.2.1 Encoding

Encoding sentences involve easy to introduce adjustable "legopieces" which we can optimize for end-to-end performance. And so, we have the context or state (represented by vector s_{t-1} of size $m \times 1$), some new information (represented by vector x_t of size $d \times 1$), and the new context or state (represented by S_t of size $m \times 1$).

We represent the new vector s_t as $\tanh\left(W^{s,s} s_{t-1} + W^{s,x} x_t\right)$.

$$\textit{context or state vector} \rightarrow \underbrace{\theta}_{\substack{\uparrow \\ \textit{new information}}} \rightarrow \textit{new context or state vector}$$

We see that in a sentence encoding, the vector s_t is the summary of the previous words seen so far.

There are three differences between the encoder (unfolded RNN) and a standard feed-forward architecture

1. input is received at each layer (per word), not just at the beginning as in a typical feed-forward network
2. the number of layer varies, and depends on the length of the sentence
3. parameters of each layer (representing an applcation of RNN) are shared (same RNN at each step)

We call the θ a lego piece as it's added to each step consistently (without changing).

Note The parameter $W^{s,x}$ represents taking into account new information x_t whereas the $W^{s,s}$ represents deciding what part of the previous information to keep.

3.2.2 Gating and LSTM

The problem with the s_t representation in 3.2.1 is that we overwrite the previous informaiton compoeltely to get the new information, and it's helpful to have some control as to how much the previous information is retained or overwritten, and this is typically done by a gating network.

$$
\begin{aligned}
g_t &= \text{sigmoid}\left(W^{g,s}s_{t-1} + W^{g,x}x_t\right) \\
s_t &= (1-g_t)\odot s_{t-1} + g_t \odot \tanh\left(W^{s,s}s_{t-1} + W^{s,x}x_t\right) \\
&\underbrace{[0,1,0,0,1,1]}_{g_t} \rightarrow \underbrace{[1,0,1,1,0,0]}_{1-g_t}
\end{aligned}
$$

In an LSTM, we have a few more gates:

$$
\begin{aligned}
f_t &= \text{sigmoid}\left(W^{f,h}h_{t-1} + W^{f,x}x_t\right) && \text{(forget gate)} \\
i_t &= \text{sigmoid}\left(W^{i,h}h_{t-1} + W^{i,x}x_t\right) && \text{(input gate)} \\
o_t &= \text{sigmoid}\left(W^{o,h}h_{t-1} + W^{o,x}x_t\right) && \text{(output gate)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh\left(W^{c,h}h_{t-1} + W^{c,x}\right) && \text{(memory cell)} \\
h_t &= o_t \odot \tanh(c_t) && \text{(visible state)}
\end{aligned}
$$

To understand, instead of taking s_{t-1}, the LSTM takes the combination of two vectors, c_{t-1} and h_{t-1} which represent the context or state and the *visible* context or state, respectively.

$$
\begin{bmatrix} c_{t-1} \\ h_{t-1} \end{bmatrix} \rightarrow \underbrace{\theta}_{\uparrow x_t} \rightarrow \begin{bmatrix} c_t \\ h_t \end{bmatrix}
$$

The new information x_t gets turned into new information c_t and h_t. Instead of overwriting the previous state as seen before, c_t also takes in information of i_t and the tanh function which controls how much previous information gets to be taken forward.

Note The sign $\odot$ denotes element-wise multiplication.

3.2.3 Markov Models

We can also represent the Markov model as a feed-forward neural network (very extendable) using softmax activation function.

Let's say that we want to predict the probability of the next hot encoded vector given the previous word (encoded vector) w_{i-1}. We can then write the

probability of the kth output node as $P_k = P(w_i = k|w_{i-1})$ such that all $P_i \geq 0$ and the $\sum_k P_k = 1$.

The forward pass then becomes

$$z_k = \sum_j x_j W_{jk} + W_{0k} \in \mathbb{R} \quad P_k = \underbrace{\sigma(k)}_{\text{softmax}} = \frac{e^{z_k}}{\sum_j e^{z_j}} \implies P_j \geq 0, \sum_k P_k = 1.$$

Advantage? These NNs are quite extendable. We can, for example, take two words (hot encodings). We can also modify this by inserting hidden layer in between the two input and output layers in order to look at more complex combinations of the preceding two words in terms of how they have mapped to the probabilities.

In a trigram language model, we use the prior two words to predict the next word, and so the complexity becomes $\mathcal{O}(|v|^2 \times |v|)$ where v is the number of words and v^2 is the number of word-pairs.

3.3 Convolution Neural Network

We formally define teh convolution as an operation between 2 functions f and g:

$$(f * g)(t) \equiv \int_{-\infty}^{+\infty} f(\tau) g(t - \tau) d\tau.$$

In this integral, τ is the dummy variable for integration and t is the parameter. Intuitively, convolution 'blends' the two function f and g by expressing the amount of overlap of one function as it is shifted over another function.

Note A novel method known as vision transformers have replaced CNNs focusing on more practical projects.

4 Reinforcement Learning

- Unlike with Supervised Learning, there would typically be no labelled training dataset associated with Reinforcement Learning tasks. RL algorithms learn to pick "good" actions based on the rewards that they receive during training.
- RL is most applicable to tasks where there is no clear cut supervised training data available.
- THe goal of RL is to learn a good policy with none or limited supervision.
- Reinforcement learning algorithms can learn to take actions so as to maximize some notion of a cumulative reward instead of the reward for the next step and they can take "good" actions even without any intermediate rewards.

4.1 Reinforcement Learning I

RL Terminology

- **States** $s \in S$ (observed)
- **Action** $a \in A$
- **Transitions** $T(s, a, s') = P(s'|s, a)$ - action dependent transition probabilities T so that for each state s and action a, $\sum_{s' \in S} T(s, a, s') = 1$.
- **Reward functions** $R(s, a, s')$, representing the reward for starting in state s, taking action a and ending up in stae s' after one step. (The reward function may also depend only on s, or only s and a.)

MDPs satisfy the Markov property in that the transition probabilities and rewards depend only on the current state and action, and remain unchanged regardless of the history (i.e. past states and actions) that leads to the current state.

Utility Function In RL, the strategy of how to accumulate rewards is taken careof by utility functions. It can be understood using an analogy, take the utility of any state to be the sum of future rewards after leaving the state, or put more formally, it is the total value in rewards of the possible next states.

The main problem for MDPs is to optimize the agent's behavior. To do so, we first need to specify the criterion that we are trying to maximize in terms of accumulated rewards. We will define a utility function and maximize its expectation.

Note The utility function can be understood as the long term reward that helps us mitigate certain problems dealt with only having immediate reward.

We consider two different types of utility functions:

1. Finite horizon based utility : The utility function is the sum of rewards after acting for a fixed number n steps. For example, in the case when the rewards depend only on the states, the utility function is

$$U[s_0, s_1, \ldots, s_n] = \sum_{i=0}^{n} R(s_i) \quad \text{for some fixed number of steps } n$$

   In particular, $U([s_0, \ldots, s_{n+k}]) = U([s_0, \ldots, s_n])\, \forall k$.

2. (Infinite horizon) *discounted* reward based utility : In this setting, the reward one step into the future is discounted by a factor of γ, the reward two steps ahead by γ^2, and so on. The goal is to continue acting (without an end) while maximizing the expected discounted reward. The discounting allows us to focus on near term rewards, and control this focus by

changing γ (obv). For example, if the rewards depend only on the states, the utility function is

$$U[s_0, s_1, \ldots] = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) = \sum_{k=0}^{\infty} \gamma^k R(s_k) \leq R_{\max} \sum_{k=0}^{\infty} \gamma^k = \frac{R_{\max}}{1-\gamma}.$$

Policy For every state, $\pi^* : s \to a$ tells you the best possible action, which maximizes the expected (reward) utility.

Bellman Equations Here are a few important equations that are key to compute the MDP policy:

$V^*(s) -$ expected reward starting at s and acting optimally

$Q^*(s, a) -$ expected reward starting at s, taking action a and acting optimally

Therefore, we can write: $V^*(s) = \max_a Q^*(s, a) = Q^*(s, \pi^*(s))$

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma V(s')\right)$$

$$\implies V^*(s) = \max_a \left[\sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma V(s')\right)\right]$$

4.1.1 Value Iteration Algorithm

Let's say that $V_k^*(s)$ is the expected reward from state s after k steps. If we take k to infinity, we have $V_k^*(s) \to V(s)$. This means that $V_k^*(s)$ should tend to the optimal expected reward.

- We initialize with $V_1^*(s) = 0$.
- Iterate until $V_k^*(s) \equiv V_{k+1}^*(s) \forall s$

$$V_{k+1}^*(s) = \max_a \left[\sum_{s'} T(s, a, a') \left(R(s, a, s') + \gamma V_k^*(s')\right)\right]$$

- Compute $\pi^*(s) = \arg\max_a Q^*(s, a)$

Problem: For our RL algorithms, we would need to provide the following tuple $\langle S, A, T, R\rangle$. However, in the real world, not all of these might be directly available. We would need to have some idea of the state space for our probelm before we could even define A, T or R.

Since the goal of RL is to train an intelligent agent that can perform interesting tasks, it is possible to identify the set of actions that this agent is allowed to take. T, R might not be so readily available in the non-deterministic noisy real world.

One way is to estimate $\hat{T}, \hat{R}$ in the following manner:

$$\hat{T} = \frac{\text{count}(s, a, s')}{\sum_{s'} \text{count}(s, a, s')}$$
$$\hat{R} = \frac{\sum_{t=1} R_t(s, a, s')}{\text{count}(s, a, s')}.$$

The problem with this method is that the statisitics for $\hat{T}, \hat{R}$ for a given state cannot be collected unless the agent visits the state during the estimation process. It might lead to the agent circling around a subset of the state space leaving the rest unexplored.

For the estimates, $\hat{T}, \hat{R}$ to be reliable, we would need to collect multiple samples of them for each state. If the state space is large, then during the exploration phase, some states would be significantly less edxplored than others resulting in very noisy estimates for these states.

4.1.2 Estimating inputs for RL algorithm

Goal: estimate $\mathbb{E}[f(x)] = \sum_x p(x)f(x)$.

Model-based We sample the random variable k times:

$$x_i \sim p(x), i = 1, \ldots, k.$$

Now we calculate the likelihood of $p(x)$:

$$\hat{p}(x) = \frac{\text{count}(x)}{k}.$$

Now our estimate of the expected value becomes:

$$\mathbb{E}[f(x)] \approx \sum \hat{p}(x)f(x).$$

Model-free We sample the random variables k times:

$$x_i \sim p(x), i = 1, \ldots, k.$$

Now $\mathbb{E}[f(x)] \approx \frac{\sum_{i=1}^{k} f(x)}{k}$.

4.1.3 Q-value iteration for RL

sample_1: $R(s, a, s'_1) + \gamma \max_{a'} Q(s'_1, a')$

$\vdots$

sample_k: $R(s, a, s'_k) + \gamma \max_{a'} Q(s'_k, a')$

$Q(s, a) = \frac{1}{k} \sum_{i=1}^{k} \text{sample}_i = \frac{1}{k} \sum_{i=1}^{k} (R(s, a, s'_i) + \gamma \max_{a'} Q(s'_i, a'))$.

exponential running average $\bar{x}_n = \frac{x_n+(1-\alpha)x_{n-1}+(1-\alpha)^2x_{n-2}+\cdots}{1+(1-\alpha)+(1-\alpha)^2+\cdots}$ and so we have

$$\bar{x}_n = \alpha x_n + (1-\alpha)\bar{x}_{n-1}.$$

$$Q_{i+1}(s,a) = \alpha \cdot \text{sample} + (1-\alpha) \cdot Q_i(s,a) \quad (*)$$
$$\text{where sample is } R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

1. Initialization $Q(s,a) = 0 \forall s, a$
2. Iteration until convergence (when the difference between two values is $< \delta$)
   - Collect sample $s, a, s', R(s,a,s')$
   - $Q_{i+1}(s,a) = \alpha \cdot (R(s,a,s') + \gamma \max_{a'} Q_i(s',a')) + (1-\alpha)Q_i(s,a)$[2]

Exploration vs Exploitation Which a (action) should the agent be taking? So the question becomes- which policy? One option is for the agent to take random a's and continue that way - *exploration*. Another option is to take advantage of what the agent knows - *exploitation*. Remember that

$$\pi*(s) = \arg\max_a Q(s,a)$$

updates the policy as the agent samples, so exploitation favors policy whereas exploration doesn't care about it. Usually, we combine the two preferring exploration in the beginning of sampling and then exploitation. This method is called ϵ-*greedy* where ϵ is the probability/likelihood of taking an action randomly.

module descriptioon

[2]We can write $Q_i(s,a) + \alpha\left(R(s,a,s') + \gamma \max_{a'} Q_i(s',a') - Q_i(s,a)\right)$ as a gradient descent algorithmic equation where the α acts as the η in the SGD algorithm