Introduction to Machine Learning
Summary

Philip Hartout

March 27, 2020

Contents

1 Linear Regression

Objective, approximate:

$$\begin{aligned} f(x) &= w_1x_1 + \ldots + w_dx_d + w_0 \\ &= \sum_{i=1}^{d} w_ix_i + w_0 \\ &= \mathbf{w}^T\mathbf{x} + w_0 \end{aligned}$$

$\forall \mathbf{x}, \mathbf{w} \in \mathbb{R}^d$. This expression can be further compressed to the homogeneous representation where $\forall \tilde{\mathbf{x}}, \tilde{\mathbf{w}} \in \mathbb{R}^{d+1}$, i.e. $\tilde{x}_{d+1} = 1$. We have w.l.o.g.:

$$f(x) = \mathbf{w}^T\mathbf{x}$$

Quantify errors using residuals:

$$\begin{aligned} r_i &= y_i - f(x_i) \\ &= y_i - \mathbf{w}^T\mathbf{x_i} \end{aligned}$$

We can use squared residuals and sum over all residuals to get the cost:

$$\begin{aligned}\hat{R}(w) &= \sum_{i=1}^{n} r_i^2 && (1)\\ &= \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x_i})^2 && (2)\end{aligned}$$

Optimization objective to find optimal weight vector $\mathbf{w}$ with least squares is the following:

$$\mathbf{w} = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \left(y_i - w^T x_i\right)^2$$

1.1 Closed form solution

This can be solved in closed form:

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

where:

$$X = \begin{bmatrix} X_{1,1} & \dots & X_{1,d} \\ \vdots & \ddots & \vdots \\ X_{n,1} & \dots & X_{n,d} \end{bmatrix} \text{ and } y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

1.2 Optimization

1.2.1 Requirements

Requires a convex objective function.

Definition 1.1 (Convexity)**.** *A function is convex iff* $\forall \mathbf{x}, \mathbf{x}', \lambda \in [0,1]$ *it holds that* $f(\lambda x + (1-\lambda)x') \leq \lambda f(x) + (1-\lambda) f(x')$

Note that the least squares objective function defined in 1 is convex.

1.2.2 Gradient descent

We start with an arbitrary $w_0 \in \mathbb{R}^d$, then for $t = 0, 1, 2, \dots$ we perform the following operation:

$$w_{t+1} = w_t - \eta_t \nabla \hat{R}(w_t)$$

where η_t is the learning rate.
Under mild assumptions, if the step size is sufficiently small, the gradient descent procedure converges to a stationary point, where the gradient is zero. For convex objectives, it therefore finds the optimal solution. In the case of the squared loss and a constant step size (e.g. 0.5), the algorithm converges at linear rate. If you look at the difference in empirical value at iteration t and compare that with the optimal value, then the gap is going to shrink at linear rate. If we look for a solution within a margin ϵ, it is found in $\mathcal{O}(\ln(\frac{1}{\epsilon}))$ iterations. The fact that the objective function congerges at linear rate can be formally described as follows:

$$\exists t_0 \forall t \geq t_0, \exists \alpha < 1 \text{ s.t. } (\hat{R}(w_{t+1}) - \hat{R}(\hat{w})) \leq \alpha(\hat{R}(w_t) - \hat{R}(\hat{w}))$$

where $\hat{w}$ is the optimal value for the hyperparameters.
For computing the gradient, we recall that:

$$\nabla \hat{R}(\hat{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1}\hat{R}(w) & \dots & \frac{\partial}{\partial w_d}\hat{R}(w) \end{bmatrix}$$

In one dimension, we have that:

$$\begin{aligned}\nabla \hat{R}(w) &= \frac{d}{dw}\hat{R}(w) = \frac{d}{dw}\sum_{i=1}^{n}(y_i - w \cdot x_i)^2 \\ &= \sum_{i=1}^{n}\frac{d}{dw}(y_i - w \cdot x_i)^2 \\ &= 2(y_i - w \cdot x_i) \cdot (-x_i) \\ &= \sum_{i=1}^{n} 2(y_i - w \cdot x_i) \cdot (-x_i) \\ &= -2\sum_{i=1}^{n} r_i x_i.\end{aligned}$$

In d-dimension, we have that:

$$\nabla \hat{R}(w) = -2\sum_{i=1}^{n} r_i x_i,$$

where $r_i \in \mathbb{R}$ and $x_i \in \mathbb{R}^d$

1.2.3 Adaptive step size for gradient descent

The step size can be updates adaptively, via either:

1. **Line search**:
   Suppose at iteration t, we have w_t, $g_t = \nabla \hat{R}(w_t)$. We then define:
   $$y_t^* = \arg \min_{y \in [0,\infty)} \hat{R}(w_t) - \eta g_t$$

2. **Bold driver heuristic**:

   - If the function decreases, increase the step size.
     $$\text{If } \hat{R}(w_{t+1}) < \hat{R}(w_t) : \eta_{t+1} \leftarrow \eta_t \cdot c_{acc}$$
     where $c_{acc} > 1$
   - If the function increases, decrease the step size.
     $$\text{If } \hat{R}(w_{t+1}) > \hat{R}(w_t) : \eta_{t+1} \leftarrow \eta_t \cdot c_{dec}$$
     where $c_{dec} < 1$.

1.2.4 Tradeoff between gradient descent and closed form

Several reasons:

- Computational complexity:
  $$\hat{w} = (X^T X)^{-1}(X^T y)$$
  $(X^T X)$ can be computed in $\mathcal{O}(nd^2)$, $(X^T X)^{-1}$ can be computed in $\mathcal{O}(d^3)$.
  By comparison, for gradient descent calculating $\nabla \hat{R}(w) = \sum_{i=1}^{n}(y_i - w^T x_i)x_i$ can be computed in $\mathcal{O}(nd)$, where $n = \ln(\frac{1}{\epsilon})$

- the problem may not require an optimal solution.

- many problems do not admit a closed form solution.

1.3 other loss functions

Least squares is part of a general case of the following general loss function, which is convex for $p \geq 1$.

$$l_p(r) = |r|^p \tag{3}$$

Least squares is where $p = 2$.

2 Probability (interlude)

2.1 Gaussians

The p.d.f. of a Gaussian distribution is given by:

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x-\mu}{2\sigma^2}\right) \tag{4}$$

The p.d.f. of a multivariate Gaussian distribution is given by:

$$\frac{1}{2\pi\sqrt{|\sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\sigma^{-1}(x-\mu)\right) \tag{5}$$

where:

$$\sigma = \begin{pmatrix} \sigma_1^2, \sigma_{12} \\ \sigma_{21}, \sigma_2^2 \end{pmatrix} \text{ and } \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} \tag{6}$$

2.2 Expectations

Expected value of a random variable can be calculated as follows:

$$\mathbb{E} = \begin{cases} \sum_x xp(x) & \text{if } X \text{ is discrete} \\ \int xp(x)dx & \text{if } X \text{ is continuous} \end{cases}$$

Expectations respect linear properties, i.e. let X, Y be random variable and $a, b \in \mathbb{R}$, then we have $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$.

3 Generalization and model validation

3.1 Fitting nonlinear functions via linear regression

Using nonlinear features of our data (basis functions), we can fit nonlinear functions via linear regression. Then, the model takes on the form:

$$f(\mathbf{x}) = \sum_{i=1}^{d} w_i\phi(\mathbf{x}) \tag{7}$$

where $\mathbf{x} \in \mathbb{R}^d$, $x \mapsto \tilde{x} = \phi(\mathbf{x}) \in \mathbb{R}^d$ and $w \in \mathbb{R}^d$.

- 1 dim.: $\phi(\mathbf{x}) = \left[1, x, x^2, \ldots, x^k\right]$
- 2 dim.: $\phi(\mathbf{x}) = \left[1, x_1, x_2, x_1^2, x_2^2, \ldots, x_1^k, x_2^k\right]$
- p dim.: $\phi(\mathbf{x})$ vector of all monomials in $x_1, \ldots, x_p$ of degree up to k.

3.2 Achieving generalization

3.2.1 Independence and identical distribution

A fundamental assumption needs to be met: the dataset is generated from an independently and identically distributed from some unknown distribution P, i.e:

$$(x_i, y_i) \sim P(\mathbf{X}, Y).$$

The i.i.d. assumption is invalid when:

- we deal with time series data
- spatially correlated data
- correlated noise

If violated, we can still use ML but the interpretation of the results needs to be carefully analyzed. The most important thing is to choose the train/test split to assess the desired generalization properties of the trained model.

3.3 Expected error and generalization error

Once the iid assumption is verified, our goal is then to minimize the expected error (true risk) under P, i.e.:

$$\begin{aligned} R(\mathbf{w}) &= \int P(\mathbf{x}, y)(y - \mathbf{w}^T\mathbf{x})^2 dxdy \\ &= \mathbb{E}[(y - \mathbf{w}^T\mathbf{x})^2] \end{aligned}$$

The true risk can be estimated by the empirical risk on a sample dataset D:

$$\hat{R}_D(\mathbf{w}) = \frac{1}{|D|} \sum_{\mathbf{x}, y \in D} (y - \mathbf{w}^T\mathbf{x})^2$$

The reason behind this approximation is because of the law of large numbers

Definition 3.1 (Law of large numbers). $\hat{R}_D(\mathbf{w}) \to R_D(\mathbf{w})$ *for any fixed* $\mathbf{w}$ *as* $|D| \to \infty$.

$$\hat{\mathbf{w}}_D = \arg\min_{\mathbf{w}} \hat{R}_D(\mathbf{w}) \tag{8}$$

$$\mathbf{w}* = \arg\min_{\mathbf{w}} \hat{R}(\mathbf{w}) \tag{9}$$

We don't want to minimize the empirical risk given in equation 8 but the true risk given in equation 9, which are similar as the amount points in the dataset increases.

3.4 Uniform convergence

For learning via empirical risk minimization, uniform convergence is required, i.e.:

$$\sup_{\mathbf{w}} |R(\mathbf{w}) - \hat{R}_D(\mathbf{w})| \to 0 \text{ as } |D| \to \infty$$

Note that this is not implied by the law of large numbers alone, but depends on model class. It holds for instance for squared loss on data distributions with bouded support. Statistical learning theory is required to define these properties.

3.5 Evaluation of performance on training data

In general in holds that:

$$\mathbb{E}_D[\hat{R}_D(\hat{\mathbf{w}}_D)] \leq \mathbb{E}_D[R_D(\hat{\mathbf{w}}_D)]$$

Proof.

$$\begin{aligned}
\mathbb{E}[\hat{R}_D(\hat{\mathbf{w}}_D)] &= \mathbb{E}_D[\min_{\mathbf{w}} \hat{R}_D(\mathbf{w})] && \text{(ERM)}\\
&\leq \min_{\mathbf{w}} \mathbb{E}_D[\hat{R}_D(\mathbf{w})] && \text{(Jensen's inequality)}\\
&= \min_{\mathbf{w}} \mathbb{E}_D[\frac{1}{|D|}\sum_{i=1}^{|D|}(y_i - wx_i)^2] && \text{(Definition of } \hat{R}_D(.))\\
&= \min_{\mathbf{w}} \mathbb{E}_D[\frac{1}{|D|}\sum_{i=1}^{|D|}(y_i - wx_i)^2] && \text{(linear expectations)}\\
&= \min_{\mathbf{w}} R(\mathbf{w}) \leq \mathbb{E}[R(\hat{w}_D)]
\end{aligned}$$

□

Thus, we obtain an overly optimistic estimate. A more realistic evaluation would be to use a separate test set from the same distribution P. Then:

- Optimize w on training set:
$$\hat{\mathbf{w}}_{D_{\text{train}}} = \arg\min_{\approx} \hat{R}_{\text{train}}(\hat{\approx})$$
- Evaluate on test set:
$$\hat{R}_{\text{test}}(\hat{\mathbf{w}}) = \frac{1}{|D_{\text{test}}|}\sum_{\mathbf{x},y\in D_{\text{test}}}(y - \hat{\mathbf{w}}^T\mathbf{x})^2$$
- Then:
$$\mathbb{E}_{D_{\text{train}},D_{\text{test}}}[\hat{R}_{D_{\text{test}}}(\hat{\mathbf{w}}_{D_{\text{train}}})] = \mathbb{E}_{D_{\text{train}}}[R(\hat{\mathbf{w}}_{D_{\text{train}}})]$$

Proof. Let $D_train = D$, $D_test = V$ and $D, V \sim P$. Then:

$$\begin{aligned}
\mathbb{E}_{D,V}[\hat{R}_V(\hat{\mathbf{w}}_D)] &= \mathbb{E}_D[\mathbb{E}_V[\hat{R}_V(\hat{\mathbf{w}}_D)]] && \text{independence of } D, V\\
&= \mathbb{E}_D[\mathbb{E}_V[\frac{1}{|V|}\sum_{i=1}^{|V|}(y_i - \hat{\mathbf{w}}_D^T x_i)^2]] && \text{(Definition of } \hat{R}_D(.))\\
&= \mathbb{E}_D[\frac{1}{|V|}\sum_{i=1}^{|V|}\mathbb{E}_{x_i,y_i}(y_i - \hat{\mathbf{w}}_D^T x_i)^2] && \text{since } (x_i, y_i)\perp D\\
&= \mathbb{E}_D[R(\hat{\mathbf{w}}_D)]
\end{aligned}$$

□

3.6 Evaluation for model selection

For each candidate model m, we repeat the following procedure for $i = 1 : k$:

- We split the same dataset into training and validation sets:
$$D = D_{\text{train}}^{(i)} \biguplus D_{\text{val}}^{(i)}$$
- We train the model:
$$\hat{\approx}_[i, m] = \arg\min_{\mathbf{w}} \hat{R}_{\text{train}}^{(i)}(\mathbf{w})$$

- Then we estimate the error:

$$\hat{R}_m^{(i)} = \hat{R}_{\text{val}}^{(i)}(\hat{\mathbf{w}}_i)$$

Finally, select the model:

$$\hat{m} = \arg\min_m \frac{1}{k} \sum_{i=1}^{k} \hat{R}_m^{(i)}$$

3.7 Splitting the data for model selection

This splitting can be done randomly through Monte Carlo cross-validation.

- Pick training set of given size uniformly at random
- Validate on remaining points
- Estimate prediction error by averaging the validation error over multiple random trials.

It can also be achieved through k-fold cross-validation, which is the default choice.

- Partition the data into k folds
- Train on $k-1$ folds, evaluating on remaining fold.
- Estimate prediction error by averaging the validation error obtained while varying the validation fold.

Note that the cross-validation error is almost unbiased for large enough k. The following should be considered to pick k:

- Too small:
  - Risk of overfitting on test set
  - Using too little data for training
  - Risk of underfitting to training set
- Too large:
  - In general, leads to better performance. $k = n$ is perfectly fine, specific instance called leave-one-out cross-validation
  - Higher computational complexity.

In practice, $k = 5$ or $k = 10$ is ofen used and works well.

3.8 Best practice for evaludating models in supervised learning

Follow the following steps:

- Split data set into training and test set
- Never look at test set when fitting the model. For example, use k-fold cross-validation on training set
- Report final accuracy on test set, but never optimize on it.

Note that this procedure only works if the data is i.i.d. I.e. one should be careful if there are temporal trends or other dependencies.

4 Regularization

We want to avoid having overly complex models when minimizing the loss function. This can be achieved through regularization, which encourages small weights via penalty functions, which are called regularizers.

4.1 Ridge regression

This is a regularized optimization problem:

$$\min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T x_i)^2 + \lambda \|\mathbf{w}\|_2^2 = \sum_{j=1}^{d} \mathbf{w}_j^2 \qquad \forall \lambda \geq 0$$

4.1.1 Closed form solution

This can be optimized using the closed form solution or gradient descent. The closed form for Ridge regression is:

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X} + \lambda \mathbf{R})^{-1}\mathbf{X}^T y$$

where $I \in \mathbb{R}^{d\times d}$ is the identity matrix.

4.1.2 Gradient descent

$$\nabla \left(\frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \|\mathbf{w}\|_2^2 \right) = \nabla_{\mathbf{w}}\hat{R}(\mathbf{w}) + \lambda \nabla_{\mathbf{w}} \|\mathbf{w}\|_2^2$$

One step of the gradient descent is therefore performed as follows:

$$\begin{aligned} w_{t+1} &\leftarrow w_t - \eta_t(\nabla_{\mathbf{w}}\hat{R}(\mathbf{w}_t) + 2\lambda \mathbf{w}_t) \\ &= (1 - 2\lambda\eta_t)\mathbf{w}_t - \eta_t \nabla_{\mathbf{w}}\hat{R}(\mathbf{w}_t) \end{aligned}$$

Choosing the regularization parameter is done through cross-validation. Typically, the choice is between values of λ values which are logarithmically spaced.

4.1.3 Generalization of a tradeoff in ML

A lot of supervised learning problems can be written in this way:

$$\min_{\mathbf{w}} \hat{R}(\mathbf{w}) + \lambda C(\mathbf{w}).$$

It's possible to control complexity by varying regulzation parameter λ.

4.2 Renormalizing data through standardization

This process ensures that each feature has zero mean and unit variance:

$$\tilde{x}_{i,j} = \frac{(x_{i,j}) - \hat{\mu}_j}{\hat{\sigma}}_j$$

where $x_{i,j}$ is the value of the j^{th} feature of the i^{th} data point:

$$\hat{\mu}_j = \frac{1}{n}\sum_{i=1}^{n} x_{i,j} \qquad \sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(x_{i,j} - \hat{\mu}_j)^2$$

5 Classification

Definition 5.1 (Classification)**.** *Classification is an instance of supervised learning where Y is discrete (categorical). We wanto to assign data points X (documents, queries, images, user visits) a label Y (spam/not spam, topic such as sports, politics, entertainment, click/no-click etc).*

The input of the model is labeled data set with positive and negative examples, see Figure 1. The output is a decision rule, i.e. a hypothesis. Given a dataset $D = \{(x_1, y_1), \ldots (x_n, y_n)\}$, we have:

$$y \approx h_{\mathbf{w}}(x) = \text{sign}(w^T x)$$

Linear classification works well in high-dimensional settings when using the right features; prediction is typically very efficient despite linear classification seeming very restrictive at first.

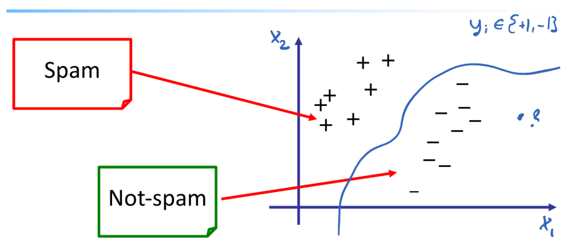

Figure 1: Illustration of binary classification

5.1 Finding linear separators

Writing the search for a classifier can be seen as an optimization problem: we seek the set of weights $\mathbf{w}$ that minimizes the number of mistakes, i.e.:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}\in\mathbb{R}} \sum_{i=1}^{n} [y_i \neq \text{sign}(w^T x_i)]$$

$$= \begin{cases} 1 \text{ if } y_i \neq \text{sign}(w^T x_i) \\ 0 \text{ otherwise.} \end{cases}$$

The goal is then to optimize the following function:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}\in\mathbb{R}} \frac{1}{n} \sum_{i=1}^{n} [y_i \neq \text{sign}(w^T x_i)]$$

$$= \frac{1}{n} \sum_{i=1}^{n} \updownarrow(w; x_i, y_i)$$

Note that this poses as challenge as it is not convex or even differentiable. Therefore we need to replace this loss by a tractable loss function for the sake of optimization/model fitting. When evaluating a model, we then use the original cost/performance function. The function we can use to optimize in this case is the surrogate loss:

$$l_P(\mathbf{w}; y_i, x_i) = \max(0, -y_i \mathbf{w}^T x_i)$$

which is also referred to as the perceptron loss.

The gradient of the perceptron loss function can be computed as follows:

$$R(\hat{w}) = \sum_{i=1}^{n} \max(0, -y_i w^T x_i)$$

$$\nabla R(\hat{w}) = \sum_{i=1}^{n} \nabla_{\mathbf{w}} \max(0, -y_i w^T x_i)$$

$$= \begin{cases} 0 \text{ if } y_i w^T x_i \geq 0 \text{ i.e. correctly classified } & -y_i x_i \text{ otherwise} \end{cases}$$

So we have the following update rule:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w} w_t + \eta_t \sum_{i:(x_i,y_i) incorrectly classified by w}^{x_i y_i}$$

5.2 Stochastic gradient descent

Computing the gradient requires summing over all data, which is inefficient for large datasets. Additionally, our initial estimates are likely very wrong and we can get a good unbiased gradient estimate by evaluating the gradient on few points. In the worst case, we can evaluate only one randomly chosen point, which is a procedure called stochastic gradient descent. It consists of the following steps:

1. Start at an arbitrary $\mathbf{w}_0 \in \mathbb{R}^d$
2. For $t = 1, 2, \dots$ do:
   - Pick data point $(\mathbf{x}', y') \in D$ from training set uniformly at random (with replacement), and set:
     $$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla \updownarrow(\mathbf{w}_t; \mathbf{x}', y')$$

Where η_t is called the learning rate. Guaranteed to converge under mild conditions, if:

$$\sum_t \eta_t = \infty \text{ and } \sum_t \eta_t^2 < \infty$$

for instance $\eta_t = \frac{1}{t}$ and $\eta_t = \min(c, \frac{c'}{t})$.
The perceptron algorithm is just stochastic gradient descent on the perceptron loss function $\updownarrow_P$ with learning rate 1.

Theorem 1 (Perceptron algorithm). *If the data is linearly separable, the perceptron will obtain a linear separator.*

The variance of the gradient estimate can be reduced by averaging over the gradients w.r.t. multiple randomly selected points, which are called minibatches. Adaptive learning rates can be additionally applie. There exist various approaches for adaptively tuning the learning rate. Often times, these even use a different learning rate per feature. Examples of adaptive learning rate algorithms include AdaGrad, RMSProp, Adam, ...

5.3 Hinge loss vs. perceptron loss

The Hinge loss encourages the margin of the classifier, and is defined as follows:

$$\updownarrow_H(\mathbf{w}; \mathbf{x}, y) = \max\left\{0, 1 - y\mathbf{w}^T\mathbf{x}\right\}$$

5.4 Support vector machines

The optimization objective for the support vector machine is defined as minimizing Hinge loss while also adding another regularization term. There are several lines that need to be considered in the max. margin linear classification, which are summarised in figure 2
Support vector machinea re widely used, very effective linear classifiers. They behave almost like a perceptron. The only differences include:

- Optimize slightly different, shifted loss (hinge loss)
- They regularize the weights

It can be optimized using a stochastic gradient descent. A safe choice for the learning rate is:

$$\eta_t = \frac{1}{\lambda t}$$

5.4.1 Stochastic gradient descent for support vector machines

Let's recall the ojective function:

$$\hat{R}(\mathbf{w}) = \sum_{i=1}^{n} \updownarrow_H(\mathbf{w}_i; x_i, y_i) + \lambda \|[\| \mathbf{w}_2^2$$

This requires taking care of the regularizer, as follows:

$$\hat{R}(\mathbf{w}) = \sum_{i=1}^{n} \left(\updownarrow_H(\mathbf{w}_i; x_i, y_i) + \frac{\lambda}{n} \|[\| \mathbf{w}_2^2 \right) = f_i(\mathbf{w})$$

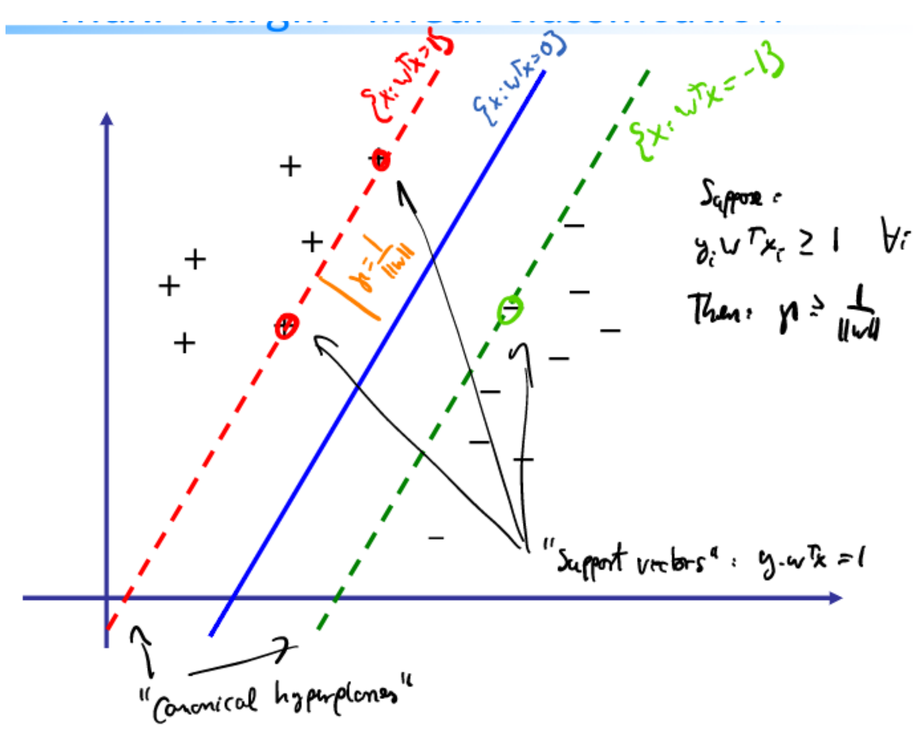
$\{x: w^Tx=1\}$
$\{x: w^Tx=0\}$
$\{x: w^Tx=-1\}$
$\gamma = \frac{1}{\|w\|}$
Suppose: $y_i w^T x_i \geq 1 \quad \forall i$
Then: $\gamma \geq \frac{1}{\|w\|}$
"Support vectors": $y_i w^T x = 1$
"Canonical hyperplanes"

Figure 2: Important elements and their defining equations for support vector machines.

So, in order to estimate the gradient $\nabla_{\mathbf{w}}\hat{R}(\mathbf{w})$ which is equivalent to:

$$\nabla_{\mathbf{w}}\hat{R}(\mathbf{w}) = \sum_{i=1}^{n} \nabla_{\mathbf{w}} f_i(\mathbf{w})$$

where:

$$\nabla_{\mathbf{w}} f_i(\mathbf{w}) = \nabla \updownarrow_H (\mathbf{w} + \frac{\lambda}{n} \nabla \|\mathbf{w}\|_2^2)$$

so the gradient of the regularization term is just $2\mathbf{w}$. For the Hinge loss:

$$\nabla \updownarrow_H = \nabla \max(0, 1 - y_i \mathbf{w}^T x_i) = \begin{cases} 0 \text{ if } y_i w^T x_i \geq 1 \\ -y_i x_i \text{ otherwise.} \end{cases}$$

Therefore the entire update rule for stochastic gradient descent for SVM is:

$$\mathbf{w}_{t+1} \leftarrow w_t \left(1 - \eta_t \frac{2\lambda}{n}\right) + [y_i w^T x_i]$$

The regularization parameter can be picked via cross-validation just like in linear regression. Instead of using the Hinge loss for validation, the target performance metric needs to be used.

5.5 Key takeaways

The key takeaways are:

- The perceptron is an algorithm for linear classification
- It applies SGD on the perceptron loss
- Mini-batches exploit parallelism and reduce variance compared to a single sample
- The perceptron loss is a convex surrogate function for the 0-1 misclassification loss

- It is guaranteed to produce a feasible solution if the data is separable
- SGD is much more generally applicable
- SVMs are closely related to Perceptron, they use a hinge loss and regularization.

Summary so far:

- List of represnetations/features: linear hypotheses, nonlinear hypotheses with nonlinear feature transforms.
- Model/Objective: loss function (squared loss, 0/1 loss, perceptron loss, Hinge loss) + regularization (L^2 norm)
- Method: exact solution, gradient descent, mini-batch SGD, convex programming.
- Evaluation metric: MSE, accuracy
- Model selection: k-fold cross-validation, Monte Carlo CV.

6 Feature selection

Reasons why we don't want to work with all potentially available features:

- interpretability: understand which features are most important
- generalizationL simpler models may generalize better
- storage computation and cost: if we select the most important features we don't need to store, sum and acquire data for unused features.

The naive way to select features is to try all subsets and pick the best features via cross-validation. Greedy feature selection, which is a general purpose approach, consists of greedily add or remove features to maximize cross-validated prediction accuracy and mutual information or other notions of informativeness not discussed here. It can be used for any method, not only linear regression or classifiers.

6.1 General greedy approach

Consider the set of features $V = \{1, \dots, d\}$. We then define the cost function for scoring subsets S of V. $\hat{L}(S)$ is the cross-validation error using features in S only. More precisely:

$$\mathbf{x}_i = [x_{i,1}, \dots, x_{i,d}] \to \mathbf{x}_{S,i} = [x_{i,j}, \dots, x_{i,j_k}]$$

where S is defined as:

$$S = \{j, \dots, j_k\}, k = |k|$$

We then train the model on $\{x_{j,1}, y_1, \dots, x_{j,n}, y_n\}$, in order to obtain an estimate based on the weights $\hat{\mathbf{w}}_S$ with associated loss $\hat{L}(S)$, which represents the cross-validated performance of the weight estimates $\hat{\mathbf{w}}_S$.

6.2 Greedy forward selection

Start with $S = \emptyset$ and $E_0 = \infty$, then for $i = 1 : d$, we find the best element to add, i.e.:

$$s_i = \arg\min j \in V \setminus S\hat{L}()S \cup \{j\}$$

then we compute the error:

$$E_i = \hat{L}(S \cup \{s_i\})$$

if $E_i > E_{i-1}$ break, else set $S \leftarrow \cup \{s_i\}$

6.3 Greedy backward selection

Start with $S = V$ and $E_{d+1} = \infty$, then for $i = d : -1 : 1$, we find the best element to remove, i.e.:

$$s_i = \arg\min j \in S\hat{L}(S \setminus \{j\})$$

then we compute the error:

$$E_i = \hat{L}(S \setminus \{s_i\})$$

if $E_i > E_{i-1}$ break, else set $S \leftarrow \setminus \{s_i\}$

6.4 Advantages and drawbacks of forward vs. backward feature selection

	Forward Feature Selection	Backward Feature Selection
Method Advantages	Faster (if few relevant features)	Can handle "dependent" features
Method drawback	Computational cost	Computational cost
	Suboptimal	Suboptimal

We want a method that simultaneously solves the learning and the feature selection problem via a single optimization step. So far we have only done optimization via sparsity: i.e. explicitely select a subset of features. This is equivalent to constraining $\mathbf{w}$ to be sparse, i.e. contain at most k non-zero entries. Alternatively, we can penalize the number of nonzero entries:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - w^T x_i)^2 + \lambda \|w\|_0$$

However, this is a difficult combinatorial optimization problem. The key idea then is to replace $\|\mathbf{w}\|_0$ by a more tractable term.

The idea here is to use L_1 as a surrogate for L_0, where:

$$\|w\|_1 = \sum_{i=1}^{d} |w_i|$$

and we use $\|w\|_1$ instead of $\|w\|_0$.

6.5 Lasso regression

In ridge regression, we use $\|\mathbf{w}\|_2^2$ to control the weights. In Lasso, we replace $\|\mathbf{w}\|_2^2$ by $\|\mathbf{w}\|_1$, hence leading to the following L1-regularized regression:

$$\min_{\mathbf{w}} \lambda \|w\|_1 + \sum_{i=1}^{n} (y_i - \mathbf{w}^T x_i)^2$$

This alternative penalty encourages coefficients to be exactly 0, which entails an automatic feature selection.

The regularization parameter can be picked using cross-validation.

6.5.1 L1 regularization in SVM

The sparsity trick can be applied to SVMs as well:

$$\min_{\mathbf{w}} \|\mathbf{w}\|_1 + \sum_{i=1}^{n} \max(0, 1 - y_i \mathbf{w}^T \mathbf{x}_i)$$

This alternative penalty encourages coefficients to be exactly 0, which ignores thoses features as well, just like in Lasso regression.

6.6 Solving L1-regularized problems

The L1-norm is convex. Combined with convex losses, we can obtain convex optimization problems, which include Lasso and L1-SVM. Those problems can, in principle, be solved usng stochastic gradient descent. Convergence is, however, usually slow. and we rarely obtain exact 0 entries. Recent work in convex optimization deals with solving such problems very efficiently using proximal methods.

Method	Greedy	L1-regularization
Advantages	Applies to any prediction method	Faster (training and feature selection happen jointly)
Disadvantages	Slower (need to train many models	Only works for linear model

7 Non-linear prediction with kernels

7.1 Revisiting the perceptron/SVM

There is a fundamental observation to be made here where the optimal hyperplace lies in the span of the data.

$$\hat{\mathbf{w}} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

that means that the output of any of the models discussed so far can be rewritten as a linear combination of the feature inputs that we have seen so far. Proving this losely can be done by stating that SGD starts from 0 and constructs such a representation. A more abstract proof follows from the representer theorem.

7.2 Reformulating the perceptron

In order to make the objective function only depend on the inner product of pairs of data point and work implicitely in high-dimensional spaces as long as we can do inner products efficiently, we need to reformula the optimization problem in terms of α instead of $\mathbf{w}$. Therefore, we do the following:

$$\begin{aligned}
\hat{\mathbf{w}} &\in \arg\min_{\mathbf{w}\in\mathbb{R}^d} \sum_{i=1}^{n} \max(0, -y_i \mathbf{w}^T x_i) \text{ Note: we make the following anzats: } \hat{\mathbf{w}} = \sum_{j=1}^{n} \alpha_j y_j x_j \\
&= \sum_{i=1}^{n} \max(0, -y_i (\sum_{j=1}^{n} \alpha_j y_j x_j)^T x_i) \\
&= \sum_{i=1}^{n} \max(0, -y_i \sum_{j=1}^{n} \alpha_j y_j (x_j^T x_i)) \\
\hat{\alpha} &\in \arg\min_{\alpha\in\mathbb{R}^n} \sum_{i=1}^{n} \max(0, -c)
\end{aligned}$$

Often, computing $k(\mathbf{x}, \mathbf{x}')$ can be computed much more efficiently than $\phi(\mathbf{x})^T\phi(\mathbf{x}')$. For the polynomial kernel of degree 2, the computational complexity of computing the explicit feature map of the kernel function is in $\mathcal{O}(d^2)$ whereas it is in $\mathcal{O}(d)$ for the kernel computation.

The perceptron can then be reformulated as follows:

Algorithm 1: Kernelized perceptron

```
Result: Trained vector $\hat{\alpha}$ used for prediction
1  $\alpha_0 \leftarrow 0$;
2  for $t = 1, \dots$ do
3      Sample $(\mathbf{x}_i, y_i) \sim D$;
4      if $y_i \sum_{j=1}^{n} \alpha_j y_j k(x_j^T x_i) > 0$ then
5          $\alpha_{t+1} \leftarrow \alpha_t$;
6      else
7          $\alpha_{t+1} \leftarrow \alpha_t$;
8          $\alpha_{t+1,i} \leftarrow \alpha_{t+1,i} + \eta_t$;
9      end
10 end
```

For a new point, we can predict:

$$\hat{y} = \text{sign}(\sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x}_j, \mathbf{x}))$$

7.3 The kernel trick

Non-linear decision boundaries can be found by using non-linear transformations of the feature vectors followed by linear classification. An important aspect to keep in mind when doing these feature transformation is the dimensionality of the data that is being used. We need, for instance, $\mathcal{O}(d^k)$ dimensions to represent multivariate polynomials of degree k on d features. The challenge then becomes to efficiently implicitely operate in such high-dimensional feature spaces withough ever explicitely computing the transformation.

Definition 7.1 (The kernel trick)**.** *The kernel trick consists in expressing a problem such that it only depends on inter products, which can then be replaced by kernels.*

An example of such a kernel can be applied to the perceptron loss:

$$\hat{\alpha} = \arg\min_{\alpha_{1:n}} \frac{1}{n} \sum_{i=1}^{n} \max \left\{ 0 - \sum_{j=1}^{n} \alpha_j y_i y_j x_i^T x_j \right\}$$

$$\Leftrightarrow \hat{\alpha} = \arg\min_{\alpha_{1:n}} \frac{1}{n} \sum_{i=1}^{n} \max \left\{ 0 - \sum_{j=1}^{n} \alpha_j y_i y_j k(\mathbf{x}_j, \mathbf{x}_i) \right\}$$

7.4 The kernelized perceptron

At training time, the problem can be solved as follows:

- Initialize $\alpha_1 = \ldots = \alpha_n = 0$
- For $t = 1, 2, \ldots$
  - Pick data point (x_i, y_i) uniformly at random
  - Predict:
  $$\hat{y} = \text{sign}(\sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x}_j, \mathbf{x}_i))$$
  - If $\hat{y} \neq y_i$ set $\alpha \leftarrow \alpha_i + \eta_t$

At the time of prediction, for a new point x, we predict:

$$\hat{y} = \text{sign}(\sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x}_j, \mathbf{x}_i))$$

7.5 Kernel functions

Definition 7.2 (Kernel functions)**.** *Given a data space X, a kernel is a function $k : X \times X \to \mathbb{R}$ satisfying the following properties:*

- ***Symmetry****: for any pair of vectors $\mathbf{x}, \mathbf{x}' \in X$ it must hold that:*

$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$$

- ***Positive semi-definiteness****: for any n, any set $S = \{\mathbf{x}_1, \dots \mathbf{x}_n\} \subseteq X$, the kernel (Gram) matrix defined as:*

$$K = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & \dots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}$$

  must be positive semi-definite.

A matrix is positive semidefinite iff:

i $\forall x \in \mathbb{R}^n : \mathbf{x}^T M \mathbf{x} \geq 0$

ii All eigenvalues of $M \geq 0$

Suppose the data space $X = \{1, \dots, n\}$ is finite, and we are given a p.s.d. matrix $\mathbf{K} \in \mathbb{R}^{n \times n}$, then we can always construct a feature map:

$$\phi : Z \to \mathbb{R}^n$$

such that $\mathbf{K}_{i,j} = \phi(i)^T \phi(j)$.

Proof. $\mathbf{K}$ is p.s.d. $\Rightarrow \mathbf{K} = UDU^T$ where $D = \begin{bmatrix} \lambda_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \lambda_n \end{bmatrix}$ and $\lambda_i \geq 0 \forall i$ We then define a matrix $D = D^{1/2} T D^{1/2}$ where $D^{1/2} \begin{bmatrix} \sqrt{\lambda_1} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sqrt{\lambda_n} \end{bmatrix}$ and $\lambda_i \geq 0 \forall i$. This gives us the following: $\mathbf{K} = UD^{1/2} D^{1/2} U^T = \phi^T \phi$ where $\phi = [\phi_1 | \dots | \phi_n]$. Now it holds that taking $k(i,j) = \mathbf{K}_{i,j} = \phi_i^T \phi_j$ which means that that $\phi : X \to \mathbb{R}^n$ and $\phi : i \to \phi_i$ which is a constructed valid feature map. It shows that for finite data spaces X, positive definiteness of the function is also a sufficient condition for it being a valid kernel. □

More generally:

Theorem 2 (Mercer's theorem)**.** *Let X be a compact subset of $\mathbb{R}^n$ and $k : X \times X \to \mathbb{R}^n$ a kernel function. Then one can expand k in a uniformly convergent series of bounded functions ϕ_i s.t.*

$$k(x, x') = \sum_{i=1}^{\infty} \lambda_i \phi_i(x) \phi_i(x').$$

7.6 Examples of kernels

- Linear kernel: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$
- Polynomial kernel: $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}' + 1)^d$
- Gaussian (RBF, squared exponential kernel): $k(\mathbf{x}, \mathbf{x}') = \exp(-\left\|\mathbf{x} - \mathbf{x}'\right\|_2^2)/h^2$ where h^2 is the bandwidth/length scale parameter.
- Laplacian kernel: $k(\mathbf{x}, \mathbf{x}') = \exp(-\left\|\mathbf{x} - \mathbf{x}'\right\|_1 / h)$

7.7 Examples of non-kernels

- $k(\mathbf{x}, \mathbf{x}') = \sin(\mathbf{x})cos(\mathbf{x}')$. It is not symmetric. Take for instance $x = 0$ and $x' = \frac{\pi}{2}$
- $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T M \mathbf{x}' \forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^d, M \in \mathbb{R}^{d \times d}$

Proof. If M is symmetric: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T M^T \mathbf{x}' = \mathbf{x}'^T M^T \mathbf{x} = k(\mathbf{x}', \mathbf{x})$. If M is not symmetric, k in general is not symmetric. If M is not positive semi-definite, then the kernel function is not definite, e.g. for the situation where $M = -1$ for the normal dot product. If M is positive semi-definite: $M = UD^{\frac{1}{2}}D^{\frac{1}{2}}TU^T = V^T V$ for $V = (UD^{\frac{1}{2}})^T$ then: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T M \mathbf{x}' = \mathbf{x}^T V^T V \mathbf{x} =)(V\mathbf{x})^T (V\mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$ for $\phi(\mathbf{x}) = V\mathbf{x}$ □

7.8 Effect of kernel on function class

Given a kernel k, predictors for kernelized classification have the form:

$$\hat{y} = \text{sign}(\sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x}_j, \mathbf{x}))$$

Grafically, it looks like in figure 3.

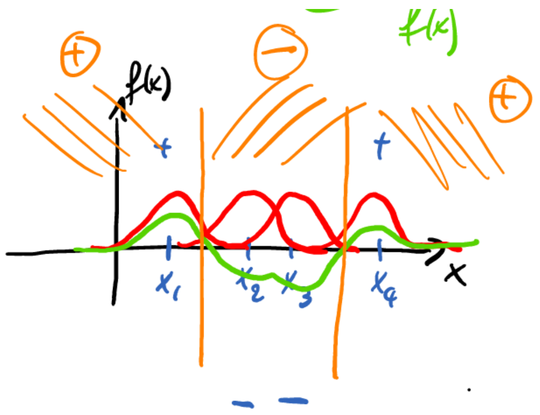

Figure 3: This figure shows that the green line is nothing but a scaled version of the indidiual Gaussian distributions and the associated decision function sign.

7.9 Graphical representations of kernels

The graphical representation of a sample gaussian and exponential kernel can be found in figures

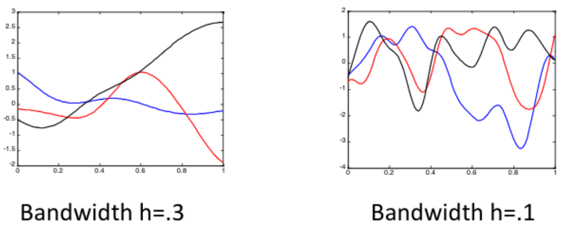

Figure 4: Graphical representation of the Gaussian kernel.

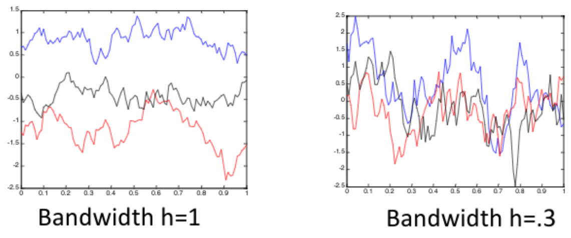

Figure 5: Graphical representation of the Laplacian kernel.

7.10 Objects where kernels can be used

The kernels can be defined on a variety of objects:

- Sequence kernels
- Graph kernels
- Diffusion kernels
- Kernels on probability distributions

Graph kernels can be used for measuring similarity between graphs by comparing random walks on both graphs. They can also be used to measure similarity among nodes in a graph via diffusion kernels not defined here.

7.11 Kernel engineering

Suppose we have two kernels:

$$k_1 : \mathcal{X} \times \mathcal{X} \to \mathbb{R} \qquad k_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

Then the following functions are valid kernels:

- $k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}')$
- $k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}')$
- $k(\mathbf{x}, \mathbf{x}') = ck_1(\mathbf{x}, \mathbf{x}')$ for $c > 0$
- $k(\mathbf{x}, \mathbf{x}') = f(k_1(\mathbf{x}, \mathbf{x}'))$ where f is a polynomial with positive coefficients or the exponential function.

7.12 The ANOVA kernel

$$k(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^{d} k_j(x_j, x_j')$$

where:

$$\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d \qquad k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \qquad k_j : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

The functions modelled by this kernel can be shown by showing what forms $f(x)$ takes:

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha_i y_i k(x^(i), \mathbf{x}) \\ &= \sum_{i=1}^{n} \alpha_i y_i \sum_{j=1}^{d} k_j(x^(i)_j, \mathbf{x}_j) \\ &= \sum_{j=1}^{d} \sum_{i=1}^{n} \alpha_i y_i k_j(x^(i)_j, \mathbf{x}_j) \\ &= \sum_{j=1}^{d} f_j(\mathbf{x}_j) \end{aligned}$$

which means that the function f decomposes into functions that depend on individual coordinates only, i.e. it is additive decomposition of the function. This property can be useful for high dimensional domains.

7.13 Modelling pairwise data

Suppose we have the following two kernels:

$$k((x,z),(x',z')) = k_x(x,x') \cdot k_z(z,z') \tag{10}$$
$$k((x,z),(x',z')) = k_x(x,x') + k_z(z,z') \tag{11}$$
$$\tag{12}$$

They can be used to represent the following data:

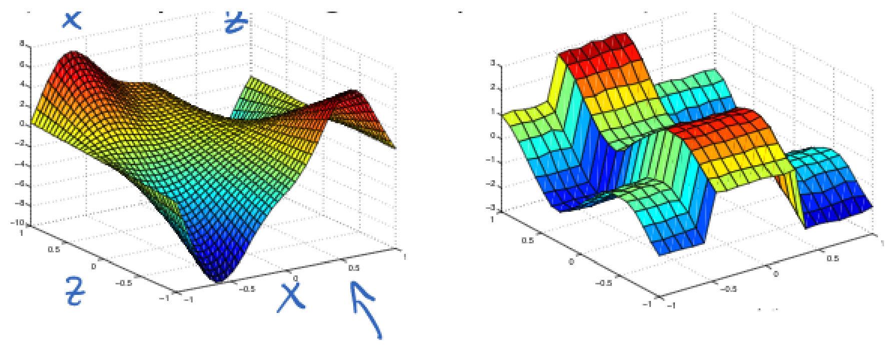

Figure 6: The left figure can be used to represent a situation where multiplying the kernel could be useful and the right figure can be used to represent a situation where adding the kernel could be useful.

7.14 Kernels as similarity functions

Kernels can be used as similarity measures. For instance, consider the Gaussian kernel $k(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2 / h^2)$. If a point $\mathbf{x}$ is close to $\mathbf{x}'$, then the value of $k(\mathbf{x}, \mathbf{x}') \approx 1$, else it is closer to 0.

7.15 Comparing k-NN to kernel perceptron

The prediction for each point in k-NN is provided by:

$$y = \text{sign}(\sum_{i=1}^{n} y_i [\mathbf{x} \text{ among } k \text{ nearest neighbors of } \mathbf{x}])$$

Method	k-NN	Kernelized perceptron
Advantages	No training required	Optimized weights can lead to improved performance, can capture global trends with suitable kernels Depends on wrongly classified examples only
Disadvantages	Depends on all data	Training requires optimisation

As we can see, it compares to the loss of the perceptron:

$$y = \text{sign}(\sum_{i=1}^{n} y_i \alpha_i k(\mathbf{x}_i, \mathbf{x}))$$

Note: choose k in k-NN using cross-validation.

7.16 Deriving non-parametric models from parametric ones

Parametric models have a finite set of paramteters. Examples of such models include linear regression, linear perceptron, etc ... Nonparametric models grow in complexity with the size of the data. They have the potential to be much more expressive but also more computationally complex. Examples of such models include the kernelized perceptron, k-NN, etc... Kernels provide a principled way of deriving non-parametrics models from parametric ones.

7.17 Kernelized SVM

The SVM optimization step can be kernelized as follows:

$$\begin{aligned}
\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, 1 - y_i \mathbf{w}^T \mathbf{x}_i\right\} + \lambda \left\|\mathbf{w}\right\|_2^2 \qquad & \text{assuming } \mathbf{w} = \sum_{j=1}^{n} \alpha_j y_j x_j \\
= \max\left\{0, 1 - y_i (\sum_{j=1}^{n} \alpha_j y_j x_j)^T x_i\right\} & \\
= \max\left\{0, 1 - y_i \sum_{j=1}^{n} \alpha_j y_j (x_j^T x_i)\right\} & \\
= \max\left\{0, 1 - y_i \sum_{j=1}^{n} \alpha_j y_j k(x_j, x_i)\right\} & \\
= \max\left\{0, 1 - y_i \alpha^T k_i\right\} &
\end{aligned}$$

where $k_i = [y_1 k(\mathbf{x}_i, \mathbf{x}_1), \ldots, y_n k(\mathbf{x}_i, \mathbf{x}_n)]^T$ and $\alpha = [\alpha_1, \ldots, \alpha_n]^T$.

The regularizer can be kernelized as follows:

$$\begin{aligned}
& \lambda \left\|\mathbf{w}\right\|_2^2 \\
=& \lambda \mathbf{w}\mathbf{w}^T = \lambda(\sum_{i=1}^{n} \alpha_i y_i x_i)^T (\sum_{j=1}^{n} \alpha_j y_j x_j) = \qquad \lambda \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_j^T x_i \\
=& \lambda \alpha^T D_y \mathbf{K} D_y \alpha
\end{aligned}$$

where $D_y = \begin{bmatrix} y_1 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & y_n \end{bmatrix}$ and $\mathbf{K} = \begin{bmatrix} k(x_1, x_1) & \ldots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \ldots & k(x_n, x_n) \end{bmatrix}$

7.18 Kernelizing linear regression

Original parametric linear regresion optimization problem is stated as follows:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (\mathbf{w}^T \mathbf{x}_i - y_i)^2 + \lambda \|\mathbf{w}\|_2^2$$

We kernelize linear regression in two parts:

$$\begin{aligned}
&= \sum_{i=1}^{n} (\mathbf{w}^T \mathbf{x}_i - y_i)^2 \\
&= \sum_{i=1}^{n} ((\sum_{j=1}^{n} \alpha_j x_j x_i)^T \mathbf{x}_i - y_i)^2 \\
&= \sum_{i=1}^{n} (\sum_{j=1}^{n} \alpha_j (x_j^T x_i) \mathbf{x}_i - y_i)^2 \\
&= \sum_{i=1}^{n} (\sum_{j=1}^{n} \alpha_j (x_j^T x_i) \mathbf{x}_i - y_i)^2 \\
&= (\alpha^2 k_i - y_i)^2
\end{aligned}$$

and

$$\begin{aligned}
&= \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j x_i^T x_j \\
&= \alpha^T \mathbf{K} \alpha
\end{aligned}$$

hence we have:

$$\hat{\alpha} = \arg \min_{\alpha \in \mathbb{R}^{\mathbb{K}}} \frac{1}{n} \sum_{i=1}^{n} (\alpha^T k_i - y_i)^2 + \lambda \alpha^T \mathbf{K} \alpha$$
$$\hat{\alpha} = \arg \min_{\alpha \in \mathbb{R}^{\mathbb{K}}} \frac{1}{n} \left\|\alpha^T \mathbf{K} - \mathbf{y}\right\|_2^2 + \lambda \alpha^T \mathbf{K} \alpha$$

which has a closed-form solution:

$$\hat{\alpha} = (\mathbf{K} + n\lambda \mathbf{I})^{-1} \mathbf{y}$$

For prediction, given a data point $\mathbf{x}$, we predict the response y as:

$$\hat{y} = \sum_{i=1}^{n} \hat{\alpha_i} k(\mathbf{x_i}, \mathbf{x})$$

7.18.1 Application: semi-parametric regression

Often, parametric models are too rigid and non-parametric models fail to extrapolate. The solution to this problem is to use additive combination of linear and non-linear kernel function as shown below:

$$k(\mathbf{x}, \mathbf{x}') = c_1 \exp\left(\|[\|]\mathbf{x} - \mathbf{x}'^2_2 / h^2 \right) + c_2 \mathbf{x}^T \mathbf{x}'$$

The decision function is then defined as follows:

$$
\begin{aligned}
f(\mathbf{x}) &= \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x}) = \sum_{i=1}^{n} \left(\alpha_i (c_i \exp(-\|[\|]\mathbf{x}_i - \mathbf{x}_2^2/h^2)) + c_2 \mathbf{x}_i \mathbf{x}'\right) \\
&= c_1 \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x}) = \sum_{i=1}^{n} \left(\alpha_i (c_1 \exp\left(\|[\|]\mathbf{x_i} - \mathbf{x}_2^2/h^2\right)) + c_2 \mathbf{x}_i^T \mathbf{x}'\right) \\
&= c_1 \sum_{i=1}^{n} \alpha_i \exp\left(-\|\mathbf{x}_i - \mathbf{x}\|_2^2/h^2\right) + \left(c_2 \sum_{i=1}^{n} \alpha_i x_i\right)^T \mathbf{x} \qquad = f_1(\mathbf{x}) + \mathbf{w}^T \mathbf{x}
\end{aligned}
$$

The efficiency of this methodology can be seen in the figure 7.

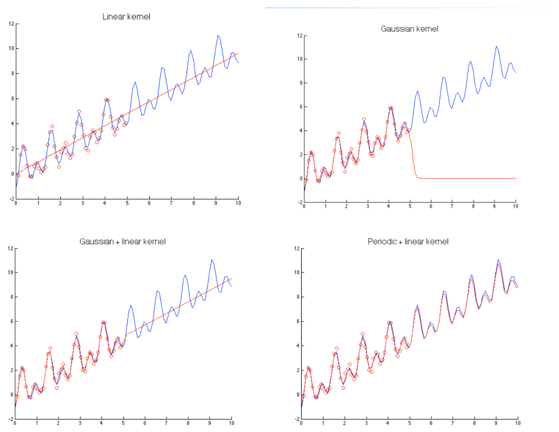

Figure 7: Various learned models overlayed with the function that we wish to learn based on the available data. Clearly, a linear kernel and a periodic parametric model seem to fit best the pattern observed.

This approach can be used to design P450 chimeras, predict protein fitness landscapes, and various other protein engineering applications.

7.19 Choosing kernels & risk of overfitting

Choosing correct kernels is a combination of domain knowledge, brute force or heuristic search using cross-validation. Since kernels map to very high dimensional spaces, it is difficult to see what we hope to be able to learn. Typically the number of parameters is drastically smaller than the number of dimensions. One way of tackling this problem is to set the number of parameters equal to that of the number of datapoints, which in this case would be called non-parametric learning. Another way of tackling this problem is to use regularization, which is built into kernelized linear regression and SVMs but not into the kernelized Perceptron. Recall the formulations of KLR and SVM:

$$
\begin{aligned}
\hat{\alpha} &= \arg\min_{\alpha} \frac{1}{n} \|[\|]\alpha^T \mathbf{K} = \mathbf{y}_2^2 + \lambda \alpha^T \mathbf{K} \alpha \\
\hat{\alpha} &= \arg\min_{\alpha} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, 1 - y_i \alpha^T \mathbf{k}_i\right\} + \lambda \alpha^T \mathbf{D_y} \mathbf{K} \mathbf{D_y} \alpha
\end{aligned}
$$

8 Class imbalance

Often data looks very imbalanced, e.g. there are more false negatives than false positives, see figure 8. Sources of imbalanced data include fraud detection datasets, spam filtering, process monitoring, medical diagnosis, feedback in recommender systems. The main issues related with imbalanced data include *performance metrics*, where the performance assessment of the model is

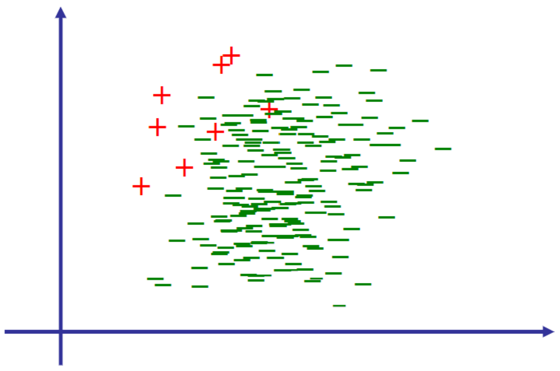

Figure 8: Example of a dataset with imbalanced data.

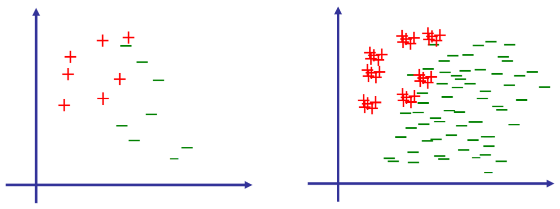

Figure 9: Illustration of subsampling (left) and upsampling (right).

not a good metric. It may be good to prefer certain mistakes over others, i.e. we trade false positives for false negatives. The minority class instance contribute little to the empirical risk. It therefore may be ignored during optimization. There are two solutions to the problem of imbalanced datasets:

- **Subsampling**: this entails removing examples from the majority class (e.g. uniformly at random) such that the resulting dataset is balanced.
- **Upsampling**: this entails repeating data points from the minority class, possibly with small random perturbation to obtain a balanced data set.
- It's also a possibility to use cost-sensitive classification methods to deal with these issues.

Illustrations of these two approaches are shown in figure 9.

8.1 Cost-sensitive classification

This entails modifying the perceptron/SVM to take class balance into account. The only different is in the cost function, where we add a class coefficient in front of the loss function:

$$\ell_{CS}(\mathbf{w};\mathbf{x},y) = c_y \ell(\mathbf{w};\mathbf{x},y)$$

For the perceptron, this becomes

$$\ell_{CS}(\mathbf{w};\mathbf{x},y) = c_y \max(0, -y\mathbf{w}^T\mathbf{x})$$

For the SVM, this becomes

$$\ell_{CS}(\mathbf{w};\mathbf{x},y) = c_y \max(0, 1 - y\mathbf{w}^T\mathbf{x})$$

where, in all cases, $c_+, c_- > 0$ control the tradeoff to be made.

An illustration of the adapted cost function can be seen in figure 10.

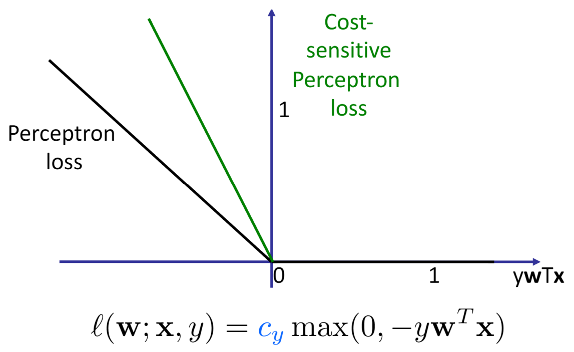

Figure 10: Perceptron loss

8.2 Avoiding redundancy

Given that the empirical loss, given by:

$$\begin{aligned}
\hat{R}(\mathbf{w}; c_+, c_-) &= \frac{1}{n}\sum_{i_{y_i=1}} c_+\ell(\mathbf{w}; x_i, y_i) + \frac{1}{n}\sum_{i_{y_i=1}} c_-\ell(\mathbf{w}; x_i, y_i) \\
\forall\alpha > 0 : \hat{R}(\mathbf{w}; \alpha c_+, \alpha c_-) &= \alpha\hat{R}(\mathbf{w}; c_+, c_-) \\
\Rightarrow \arg\min_{\mathbf{w}} \hat{R}(\mathbf{w}; \alpha c_+, \alpha c_-) &= \arg\min_{\mathbf{w}} \alpha\hat{R}(\mathbf{w}; c_+, c_-) = \arg\min_{\mathbf{w}} \alpha\hat{R}(\mathbf{w}; \frac{c_+}{c_-}, 1) \\
\Rightarrow \text{w.l.o.g.: } \alpha &= \frac{1}{c_-}.
\end{aligned}$$

Which basically means that the tradeoff to make for imbalanced datasets can be determined by one parameter.

8.3 Metrics for imbalanced datasets

A generally inappropriate metric for imbalanced datasets is accuracy, defined as

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{N}$$

Other, more suitable metrics include:

$$\begin{aligned}
\text{Precision} &= \frac{TP}{TP + FP} = \frac{TP}{P+} \\
\text{Recall} = \text{True positive rate (TPR)} &= \frac{TP}{TP + FN} = \frac{TP}{N+} \\
\text{F1 score} &= \frac{2TP}{2TP + FN + FP} = \frac{2}{\frac{1}{\text{Prec}} + \frac{1}{\text{Rec}}} \\
\text{False positive rate (FPR)} &= \frac{FP}{TN + FP}
\end{aligned}$$

An interesting observation to make is that given the probability p of observing a positive sample, we have:

$$\begin{aligned}
\mathbb{E}[\text{TPR}] &= \frac{\mathbb{E}[\text{TP}]}{n_+} = \frac{p \cdot n_+}{n_+} = p \\
\mathbb{E}[\text{TPR}] &= \frac{\mathbb{E}[\text{TP}]}{n_-} = \frac{p \cdot n_-}{n_-} = p
\end{aligned}$$

The receiver operator characteristic (ROC) curve (see figure 11) can also be used to make an assessment of a classifier.

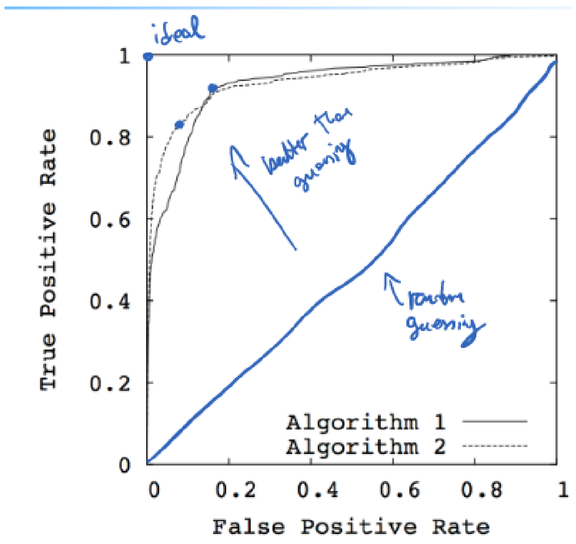

Figure 11

		Prediction outcome		
		p	**n**	**total**
actual value	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Note that a performance measure associated to the precision recall curve is the area under the curve. A random classifier has an AUC of 0.5 and an ideal classifier has an AUC of 1.s

It is also important that the confusion matrix itself is also often used to evaluate the performance of a binary or even multi-class classifier.

8.4 Obtaining the optimal tradeoff

Two ways of obtaining a good tradeoff is to either use a cost sensitive classifier and vary the tradeoff parameter. The second option is to find a single classifier and vary the classification threshold τ in the following formula:

$$y = \text{sign}(\mathbf{w}^T\mathbf{x} - \tau)$$

The optimal value for the threshold can be determined using the precision-recall curve 12.

An interesting note to make in terms of the relationship between the ROC curve and the precision recall curve is captured in the following theorem:

Theorem 3. *Algorithm 1 dominates Algorithm 2 in terms of ROC curve* $\Leftrightarrow$ *Algorithm 1 dominates Algorithm 2 in terms of precision recall curve.*

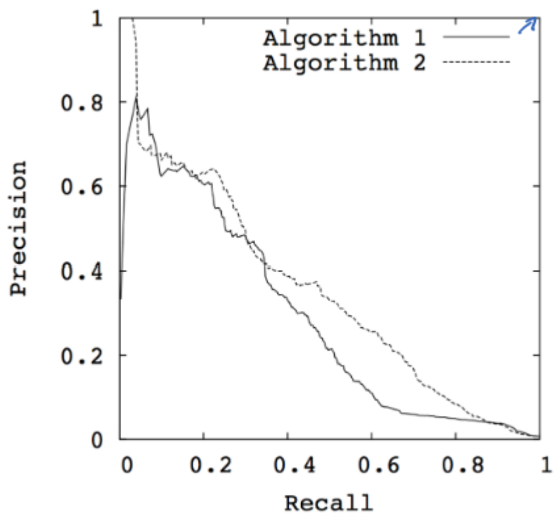

Figure 12: An example of a precision-recall curve.

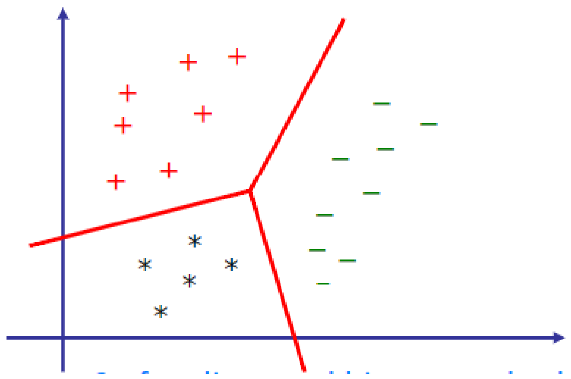

Figure 13: Illustration of a multiclass problem.

9 Multiclass problems

A multiclass problem is defined as follows. Given a dataset $\mathcal{D} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ for $y_i \in 1, \ldots, c$, we want a function $f : \mathcal{X} \to \mathcal{Y}$ where $y_i \in \mathcal{Y} = 1, \ldots, c$ and $\mathbf{x}_i \in \mathcal{X} \subseteq \mathbb{R}^d$, as illustrated in figure 13.

9.1 One-vs-all

One approach to solve the problem is to solve c binary classifiers, i.e. one for each class, where all positive samples are from class i and all negative samples represent all other points. We then classify using the classifier with the largest confidence. In other words, we fit $f^{(i)} : \mathcal{X} \to \mathbb{R}$ and $f^{(i)}(x) = \mathbf{w}^{(i)T}\mathbf{x}$ and we predict $\hat{y} = \arg\max_i f^{(i)}(\mathbf{x}) = \arg\max_i \mathbf{w}^{(i)T}\mathbf{x}$.

9.1.1 Confidence in a classification

The confidence in a classification decision can be visualized as in figure 14. For $\alpha > 0$, we have

$$\operatorname{sign}((\alpha\mathbf{w})^T\mathbf{x}) = \operatorname{sign}(\mathbf{w}^T\mathbf{x})$$

Thus although $\alpha\mathbf{w}$ and $\mathbf{w}$ implement the same decision boundary, they have a different confidence level. There are two solutions to this problem:

1. $\mathbf{w} \leftarrow \frac{\mathbf{w}}{\|\mathbf{w}\|_2}$, i.e. this is normalized to unit length.

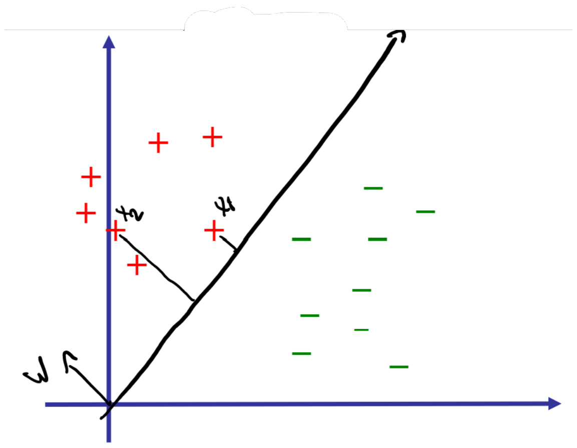

Figure 14: Illustration of the confidence in classification. We see that $\mathbf{w}^T\mathbf{x}_2 > \mathbf{w}^T\mathbf{x}_1$ so that means that the label $+$ is given with more confidence to $\mathbf{x}_2$ than for $\mathbf{x}_1$

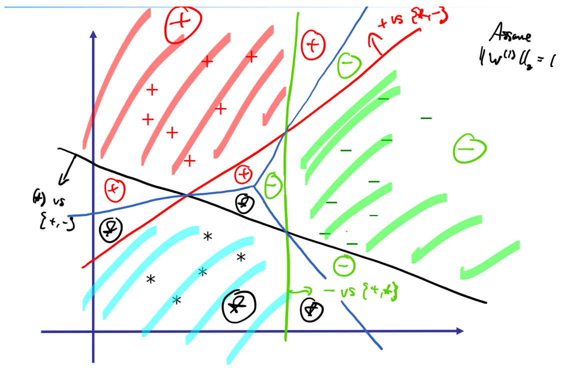

Figure 15: One-vs-all decision boundary.

2. In practice, when using regularization, the magnitude of $\|[\|]\mathbf{w}_2$ is kept under control.

The decision boundary of the one-vs-all decision can be represented as in figure 15.

9.1.2 Challenges

There are several challenges associated with the one-vs-all technique:

- It only works if classifiers produce confidence scores on the same scale.
- Individual binary classifier see imbalanced data, even if the whole data set is balanced.
- One class might not be linearly separable from all other classes.

9.2 One-vs-one

The idea is to train $c(c-1)/2$ binary classifiers, one for each pair of classes (i, j). Positive examples contain all points from class i and negative examples contain all points from class j. Then, we apply a voting scheme, i.e. the class with the highest number of positive prediction wins. A graphical representation of the one-vs-one classification is shown in figure 16.

9.3 Encodings

There are other methods to tackle multiclass problems, e.g. using other encodings (error correcting output codes). There are also other explicit multiclass models, such as the multi-class perceptron/svm etc. Some models are naturally multi-class, such as nearest neighbors, generative

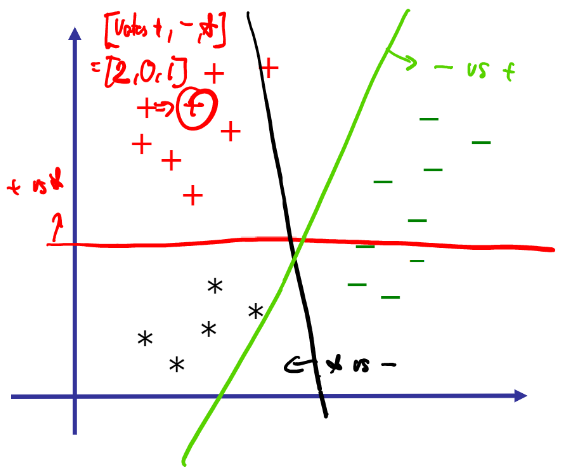

Figure 16: Illustration of the one-vs-one classification technique. Here, we see that each decision boundary is fit for each pair of classes (i, j).

Method	*One-vs-all*	*One-vs-one*
Advantages	Only c classifiers needed (faster!)	No confidence needed
Disadvantages	Requires confidence + leads to class imbalance	Need to train $c(c-1)/2$

probabilistic models, etc. However, one should keep in mind that one-vs-all/one-vs-one usually works very well.

A way to reduce the number of classes we need to work with can be reduce by using an encoding. The length of the encoding will then scale with the $log_2 c$, see table 1. Then in principle the multiclassfication problem is phrased as a decoding task of the class label, where each classifier predicts one bit, which in this case we would get away with $\mathcal{O}(\log c)$ classifiers. In this case, ideas from coding theory to do multi-class classification can be used to do multi-class classification.

9.4 Multi-class SVMs

The key idea of multi-class SVMs is to maintain c weight vectors $\mathbf{w}^1, \ldots, \mathbf{w}^c$, one for each class. Then we predict using:

$$\hat{y} \leftarrow \arg\max_{i \in 1,\ldots,c} \mathbf{w}^{(i)T}\mathbf{x}$$

Given each data point $(\mathbf{x}, y)$, we want to achieve that:

$$\mathbf{w}^{(y)}\mathbf{x} \geq \mathbf{w}^{(i)}\mathbf{x} + 1 \forall i \in \{1, \ldots, c\} \setminus y \quad (13)$$

$$\equiv \mathbf{w}^{(y)}\mathbf{x} \geq \max_{i \in \{1,\ldots,c\} \setminus \{y\}} \mathbf{w}^{(i)}\mathbf{x} + 1 \quad (14)$$

Class	*Encoding*
0	[0,0,0,0]
1	[0,0,0,1]
2	[0,0,1,0]
⋮	⋮
$c-1$	[1,1,1,1]

Table 1: Table containing the class name and corresponding encoding

9.5 Multi-class Hinge loss

$$\ell_{MC-H}(\mathbf{w}^1, \dots, \mathbf{w}^c; \mathbf{x}, y) = \max(0, 1 + \max_{j \in \{1,\dots,y-1,y+1,\dots,c\}} \mathbf{w}^{(j)T}\mathbf{x} - \mathbf{w}^{(y)T}\mathbf{x})$$

which is equal to 0 when equation 13 is satisfied.

The gradient of the Hinge loss is then defined as:

$$\nabla_{w^{(i)}} \ell_{MC-H}(\mathbf{w}^{(1:c)}; \mathbf{x}, y) = \begin{cases} 0 \text{ if equation 13 is satisfied or } i \neq y \wedge i \neq \arg\max_j \mathbf{w}^{(j)T}\mathbf{x} \\ -\mathbf{x} \text{ if 13 is not satisfied and } i = y \\ +\mathbf{x} \text{ otherwise.} \end{cases}$$

10 Neural networks

10.1 Importance and characteristics of features

Succes in learning crucially depends on the quality of features. Hand-designing features however requires domain knowledge. However, kernels could be used to engineer features as they provide a rich set of feature maps, they can fit almost any function with infinite data. However, choosing the right kernel can be challenging and the computational complexity grows exponentially with the size of the data. The question then becomes whether or not we can learn good features from the data directly. The overall objective function for neural networks to optimize is the following:

$$\mathbf{w}* = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \ell(y_i; \sum_{j=1}^{m} w_j \phi_j(\mathbf{x}_j)) = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \ell(y_i; f_i)$$

where $\ell(y_i, f_i)$ is usually defined as $\ell(y_i, f_i) = (y_i - f_i)^2$ (squared loss). A key idea then is to parametrize the feature maps, and optimize over the parameters, i.e. our optimization objective is to tune both the parameters and the weights to minimize the loss:

$$\mathbf{w}* = \arg\min_{\mathbf{w},\theta} \sum_{i=1}^{n} \ell(y_i; \sum_{j=1}^{m} w_j \phi(\mathbf{x}_i, \theta))$$

One possibility of achieving this optimization is to define the following feature map:

$$\phi(\mathbf{x}, \theta) = \varphi(\theta^T \mathbf{x})$$

where $\theta \in \mathbb{R}^d$ and $\phi : \mathbb{R} \to \mathbb{R}$ is a nonlinear function, called an *activation function.* An example of such an activation function is the sigmoid and tanh activation functions. The signmoid function is given by:

$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$

The tanh function is given by:

$$\varphi(z) = \tanh(z) = \frac{exp(z) - \exp(-z)}{exp(z) + \exp(-z)}$$

The rectified linear unit is also often used, which is defined as:

$$\phi(z) = \max(z, 0)$$

The decision function of an artificial neural network is of the form:

$$f(\mathbf{x}_i; \mathbf{w}, \theta) = \sum_{j=1}^{m} w_j \varphi(\theta_j^T \mathbf{x}) = \sum_{j=1}^{m} w_j v_j$$

More generally, the term artificial neural network refers to nonlinear functions which are nested compositions of variable linear functions composed with fixed nonlinearities.

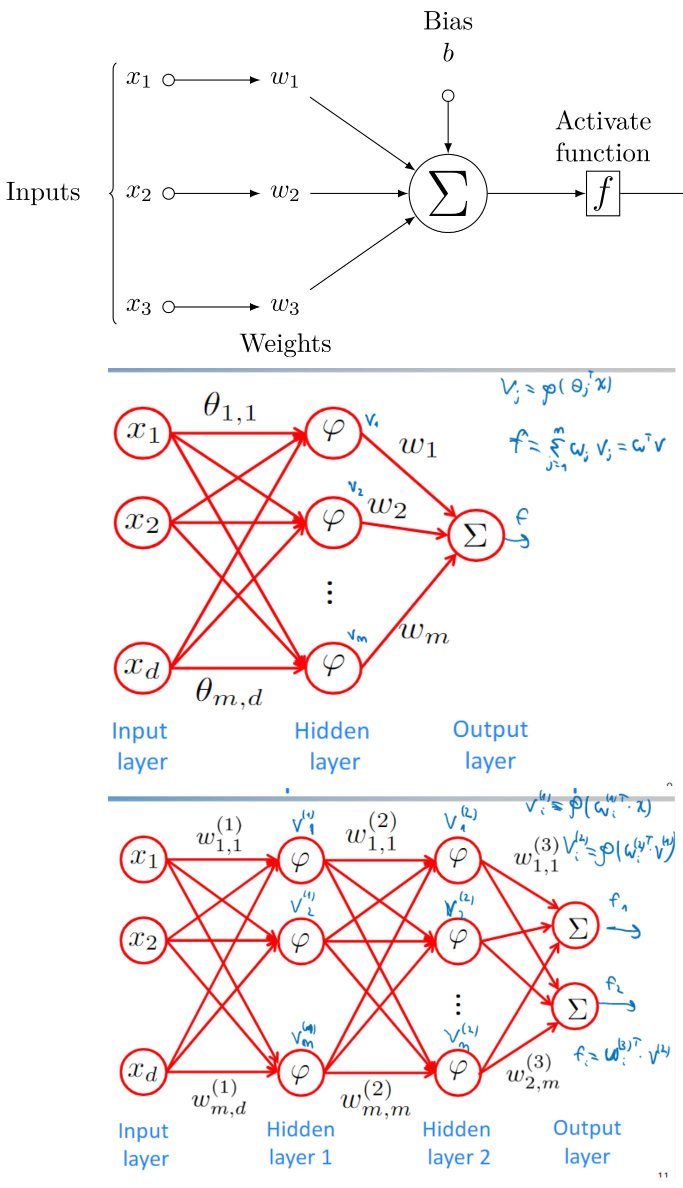

Figure 17: Graphical representations of an artificial neural network.

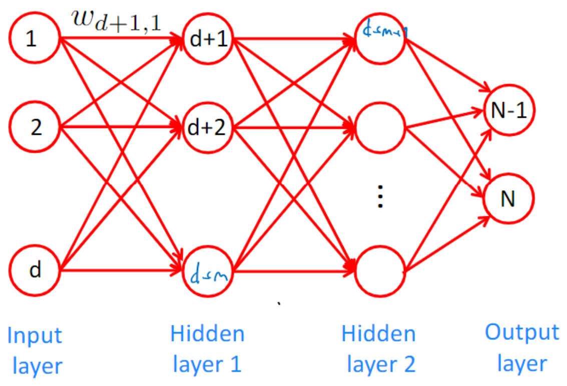

Figure 18: Indexing convention of ANNs

10.2 Graphical representation

A neural network can be represented as shown in figure 17 Note that neural networks can have multiple outputs for multi-class prediction (one output per class or multi-output regression). They can also have more than one hidden layer – networks with large amounts of neural networks with several hidden layers are called deep neural networks. Indexing in neural networks is done as shown in figure 18.

10.3 Making predictions using ANNs

Suppose we have learned all paramters $w_{i,j}$. Given an input, we now make predictions using forward propagation. The forward propagation procedure is described as follows:

1. For each unit j on the input layer, we set its value to $v_j = x_j$.
2. For each layer $\ell = 1 : L - 1$
   (a) For each unit j on layer ℓ we set its value:
   $$v_j = \varphi\left(\sum_{i \in \text{Layer}_{\ell-1}} w_{j,i} v_i\right)$$
3. For each unit j on output layer, set its value:
   $$f_j = \sum_{i \in \text{Layer}_{L-1}}^{w_{j,i} v_i}$$
4. Predict $y_j = f_j$ for regression, and $y_j = \text{sign}(f_j)$ for regression. For multiclass problems, the prediction is made as $\hat{y} = \arg\max_j f_j$.

Another way of writing the forward propagation algorithm is as follows:

1. For input layer: $\mathbf{v}^{(0)} = \mathbf{x}$
2. For each layer $\ell = 1 : L - 1$
   $$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{x}^{(\ell-1)}, \mathbf{z}^{(\ell)} \in \mathbb{R}^{m^{(d)}}$$
   $$\mathbf{v}^{(\ell)} = \varphi(\mathbf{z}^{(\ell)})$$
   where $m^{(d)}$ is the number of units in the ℓ^{th} layer. and $\varphi(\mathbf{z}^{(\ell)}) = [\varphi(\mathbf{z}_1^{(\ell)}), \varphi(\mathbf{z}_2^{(\ell)}), \dots, \varphi(\mathbf{z}_m^{(\ell)})]$.
3. For the output layer: $f = \mathbf{W}^L \mathbf{v}^{L-1}$
4. Predict $y = f$ for regression, and $y = \text{sign}(f)$ for regression. For multiclass problems, the prediction is made as $\hat{y} = \arg\max_i f_i$.

10.4 Universal approximation theorem

Neural networks are powerful modelling tools, because any decision function can be modelled by a sufficiently complex artificial neural networ'k

Theorem 4 (Universal approximation theorem)**.** *Let σ be any continuous sigmoidal function. Then finite sums of the form:*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \sigma_j)$$

are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(x)m$ of the above form, for which:

$$|G(x) - f(x)| < \epsilon \text{ for all } x \in I_n$$

10.5 Training the weights of a neural network

Given a dataset $\mathcal{D} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, we optimize the weights $\mathbf{W} = (\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)})$ y applying a loss function (e.g. the perceptron loss, the multi-class hinge loss, square loss, etc.) to output:

$$\ell(\mathbf{W}; \mathbf{y}, \mathbf{x}) = \ell(\mathbf{y}; f(\mathbf{x}, \mathbf{x})).$$

Then, we optimize the weights to minimise the loss over $\mathcal{D}$:

$$\mathbf{W}* = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{y}_i, \mathbf{x}_i)$$

When predicting multiple outputs at the same time, we usually define the loss as a sum of per-output losses as follows:

$$\ell(\mathbf{W}; \mathbf{y}, \mathbf{x}) = \sum_{i=1}^{p} \ell_i(\mathbf{W}; y_i, \mathbf{x}).$$

For regression tasks we usually use the squared loss and for classification we use the perceptron or the hinge loss.

Jointy optimizng over all weights for all layers as described in 10.5 is, in general, a non-convex optimization problem. We can nevertheless try to find a local optimum.

This local optimum can be found, for instance, using stochastic gradient descent, described as follows:

1. Initialize the weights $\mathbf{W}$.
2. For $t = 1, 2, \ldots$
   (a) Pick data point $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ uniformly at random.
   (b) Take a step in negative gradient direction:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta_t \nabla_{\mathbf{W}} \ell(\mathbf{W}, \mathbf{y}, \mathbf{x})$$

10.6 Deep learning

Generally, deep learning refers to models with nested, layered non-linearities. Common examples of deep neural networks include:

1. Classical ANN with multiple layers
2. Trained via SGD and variants
3. Some new algorithmic insights (e.g. dropout regularization) and exntensions (Convnets, resnets, RNNs, LSTMs, GRUs,...)

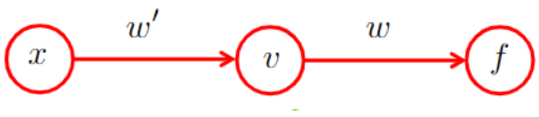

Figure 19: The computation graph of a simple ANN

Some deep learning success stories include the fact that deep neural networks achieve state of the art performance on some difficult classification tasks such as speech recognition, image recognision, natural language processing and speech translation. A lot of recent work on sequential models such as RNNs, LSTMs, GRUs,... have also proven to model data effectively.

Some crucial questions to be answered when training a deep neural network include the following:

- How can we compute the gradients?
- How should we initialize the weights?
- When should we terminate?
- How do we choose parameters (number of units/layers/activation functions/learning rate/...)?
- What about overfitting?

10.7 Computing the gradient

In order to apply SGD, we need to compute:

$$\nabla_{\mathbf{W}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$$

i.e. for each weight between any two connected units i and j, we need to compute:

$$\frac{\partial}{\partial w_{i,j}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$$

In a simple example, we have an ANN with 1 output, 1 hidden and 1 input unit: $\mathbf{W} = [\mathbf{w}, \mathbf{w}']$ as shown in figure 19. In this example we have that:

$$f(x, \mathbf{W} = w\varphi(w'x)$$

and a dataset containing 1 datapoint containing $\mathcal{D} = (x, y)$. Then the loss is defined as:

$$L(\mathbf{w}', \mathbf{w}) : L = \ell(f, y) = \ell_y(f)$$

Note that $\ell_y(f)$ is computed using forward propagation. Differentiating the loss function w.r.t. the weights gives:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{w}} &= \frac{\partial L}{\partial f}\frac{\partial f}{\partial \mathbf{w}} = \ell'_y(f) \cdot v \\ \frac{\partial L'}{\partial \mathbf{w}} &= \frac{\partial L}{\partial f}\frac{\partial f}{\partial \mathbf{v}}\frac{\partial \mathbf{v}}{\partial \mathbf{w}'} = \delta \cdot \mathbf{w} \cdot \phi'(z) \cdot \mathbf{x}. \end{aligned}$$

for instance in the case of the square loss:

$$\begin{aligned} \ell_y(f) = (f-y)^2 \\ \Rightarrow l'_y(f) = 2(f-y) \end{aligned}$$

For a slightly more complex example, as depicted in figure 20, the loss is defined as follows:

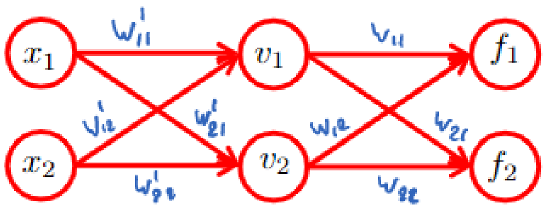

Figure 20: A more complex neural network.

$$L = \sum_{i=1}^{2} \ell_i(\sum_j w_{i,j}\varphi(\sum_k w'_{j,k}x_k)).$$

The weight vector $\mathbf{W}$ is defined as $\mathbf{W} = [(w_{i,j})_{i,j}, (w'_{i,j})_{i,j}]$. The loss is then defined as:

$$\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial f_i}\frac{\partial f_i}{\partial w_{i,j}} = \ell'_i(f_i) \cdot v_j$$

Note that $\ell'_i(f_i)$ and $\cdot v_j$ are computed using forward propagation.

$$\frac{\partial L}{\partial w'_{j,k}} = \sum_{i=1}^{2} \frac{\partial L}{\partial f_i}\frac{\partial f_i}{\partial v_j}\frac{\partial v_j}{\partial w'_{j,k}} = \sum_{i=1}^{2} \delta_i w_{i,j} \cdot \phi'(z_j) \cdot x_k$$

where $\delta_i = \frac{\partial L}{\partial f_i}$ and $\delta'_j = \delta_i w_{i,j} \cdot \phi'(z_j) \cdot x_k$

10.7.1 Derivatives of activation functions

Sigmoid:

$$\varphi(z) = \frac{1}{1+\exp(-z)}$$

$$\phi'(z) = \frac{-1}{(1+\exp(-z))^2} \cdot e^{-z} \cdot (-1) = \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} = \phi(z)(1-\phi(z))$$

The advantage of this function is that it is differentiable everywhere, however $\phi(z) \approx 0$ unless $z =\approx 0$. This leads to a problem that is called a vanishing gradient for deep models.

Rectified linear unit (ReLU):

$$\varphi(z) = \max(z, 0)$$

$$\varphi'(z) = \begin{cases} 1 \text{ if } z > 0 \\ 0 \text{ otherwise} \end{cases}$$

The disadvantage of this function is that it is not differentiable at 0, but in practice this is not a problem. A huge advantage is $\varphi'(z) = 1$ for all $z > 0$ which helps with avoiding vanishing gradients.

10.8 Backpropagation

The unvectorized version of the backpropagation algorithm is as follows:

1. For each unit j on the output layer
    - Compute the error signal $\delta_j = \ell'_j(f_j)$
    - For each unit i on layer L, compute $\frac{\partial}{\partial w_{i,j}} = \delta_j v_i$

2. For each unit j on hidden layer $\ell = L-1 : -1 : 1$
   - Compute the error signal: $\delta_j = \phi'(z_j)\sum_{i\in\text{Layer}_{\ell+1}} w_{i,j}\delta_i$
   - For each unit i on layer ℓ, compute $\frac{\partial}{\partial w_{i,k}} = \delta_j v_i$

The vectorized implementation of the backpropagation algorithm can be phrased as follows:

1. For the output layer
   - Compute the error $\delta^L = \ell'(\mathbf{f}) = [\ell'(f_!), \ldots, \ell'(f_p)]$
   - Gradient: $\nabla_{\mathbf{W}^{(L)}}\ell(\mathbf{W};\mathbf{y},\mathbf{x}) = \delta^{(L)}\mathbf{v}^{(L-1)T}$
2. For each hidden layer $\ell = L-1 : -1 : 1$
   - Compute the error $\delta^{(\ell)} = \varphi'(\mathbf{z}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)T}\delta^{(\ell+1)})$:
   - Compute the gradient: $\nabla_{\mathbf{W}^{(\ell)}}\ell(\mathbf{W};\mathbf{y},\mathbf{x}) = \delta^{(\ell)}\mathbf{v}^{(\ell-1)T}$

10.9 Initializing weights

Because the optimization problem is non-convex, the initialization of the weights matters, i.e. if we pick innapropriate weights we are more likely to converge towards a suboptimal point in the optimization landscape. Random initialization is often preferred. The rationale of this comes from the propagation of variance. In order to understand why this is the case, assume a node $v_i^{(\ell)}$ with inputs $v_1^{(\ell-1)}, \ldots, v_{n_{in}}^{(\ell-1)}$. We know that we carry out the computation $z_i^{(\ell)} = \sum_{j=1}^{n_{in}} w_{ij}v_j^{(\ell-1)} \Rightarrow v_i^{(\ell)} = \phi(z_i^{(\ell)})$ where $\phi(z) = \max(0, z)$.

10.9.1 Variance propagation

Now, we assume $\mathbb{E}[x_i] = \mathbb{E}[v_i^{(0)}] = 0$ where $v_i^{(0)}$ is the input layer and we also assume that $\text{Var}[x_i] = \text{Var}[v_i^{(0)}] = 1$. Let us further assume that the $X_1, \ldots, X_d$ are independent. Additionally, we assume that the weights are draw from a normal distribution, i.e. $w_{ij} \sim \mathcal{N}(0,1)$. Finally, we know that, supposing X, Y are independent with $\mathbb{E}[X] = 0$:

a if $\mathbb{E}[Y] = 0$, then $\text{Var}(X, Y) = \text{Var}(X)\text{Var}(Y)$

b if $\mathbb{E}[Y] \neq 0$, then $\text{Var}(X, Y) = \text{Var}(X)\mathbb{E}[Y^2]$

c For ReLU and symmetric Y: $\mathbb{E}[\phi(Y)^2] = \frac{1}{2}\text{Var}(Y)$

We then compute:

$$\begin{aligned}
\mathbb{E}[z_i^{(1)}] &= \mathbb{E}\left[\sum_{j=1}^{n_{in}} w_{i,j}x_j\right] = \sum_j \mathbb{E}[w_{ij}]\mathbb{E}[x_j] = 0 \\
\text{Var}[z_i^{(1)}] &= \sum_j \text{Var}[w_{ij}x_j] \overset{\text{a}}{=} \sum_j \text{Var}(w_{ij})\sum_j \text{Var}(x_j) = n \cdot \sigma^2 \\
\text{Var}[z_i^{(1)}] &= \sum_j \text{Var}\left[w_{ij}v_j^{(\ell-1)}\right] = \sum_j \text{Var}(w_{ij})\mathbb{E}\left[v_j^{(\ell-1)2}\right] = \frac{1}{2}n_{\text{in}}\sigma^2 := 1 \Rightarrow \sigma^2 = \frac{2}{n_{\text{in}}}
\end{aligned}$$

This means that in this setting (of course heavily idealized), that the random variables will have the same scale with the same variance across each layer.

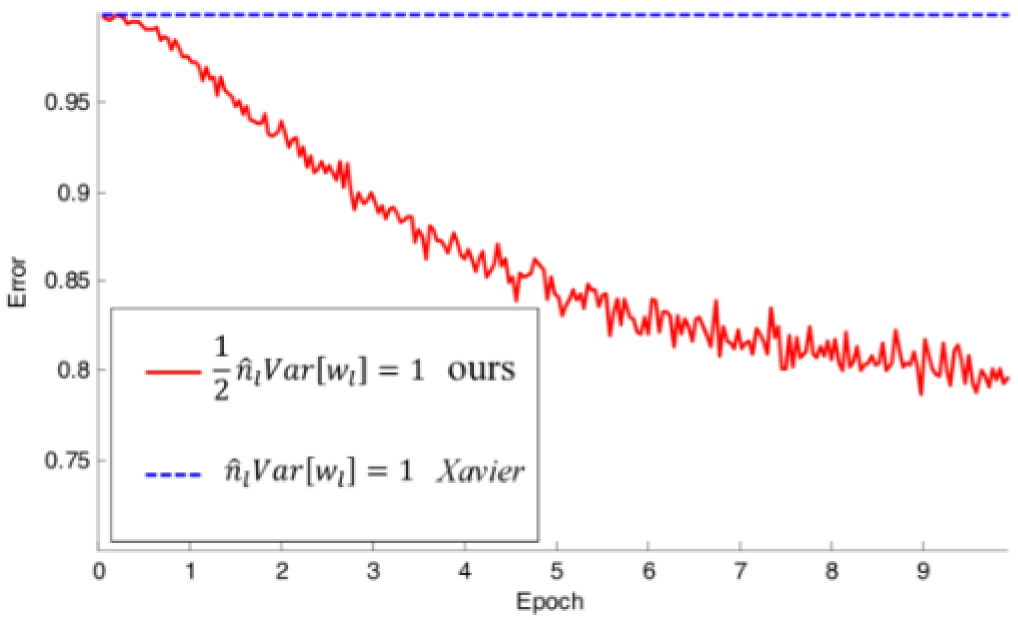

Figure 21: Weight initialization procedure and effect on training

10.9.2 Weight initialization strategies

The goal of any weight initialization procedure is to keep the variance of weights approximately constant across layers to avoid vanishing and exploding gradients. Usually, random initialization works well, for instance:

- Glorot initialization using tanh:

$$w_{i,j} \sim \mathcal{N}(0, 1/(n_{in}))$$
$$w_{i,j} \sim \mathcal{N}(0, 1/(n_{in} + n_{out}))$$

- He (ReLU):

$$w_{i,j} \sim \mathcal{N}(0, 2/n_{in})$$

The effect of improper initialization strategies is that the weights do not converge over time, as shown in figure 21.

10.10 Learning rate

To implement the SGD rule, a learning rate needs to be chosen. Usually, its a good idea to start with a fixed small learing rate, and decrease slowly after some iteration, e.g.:

$$\eta_t = \min(0.1, 100/t)$$

Often, a piecewise constant learning rate schedules the drop in learning rate, e.g. drop after some number of epochs.

Learning with momentum is a common extension to training with SGD as it can help to escale local minima. The idea behind this approach is to not only move into the direction of the gradient, but also in direction of the last weight update. In this case the weight update looks like this:

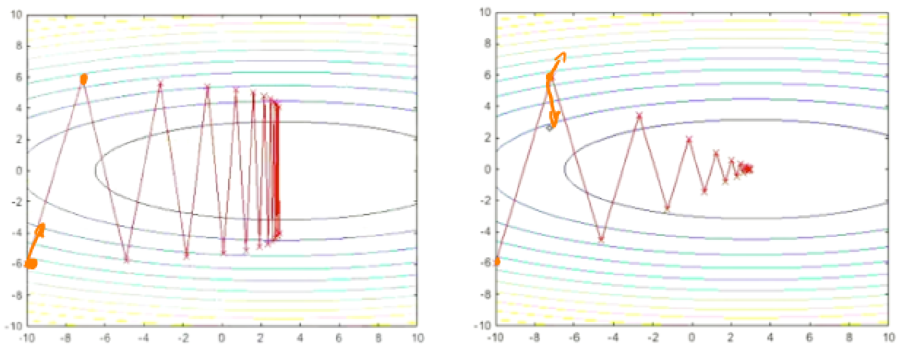

Figure 22: Oscillations of the optimization function.

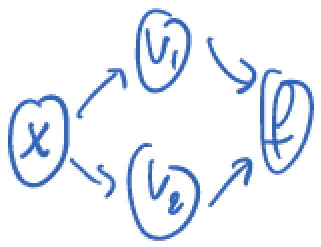

Figure 23: Network topology used to illustrate weight-space symmetries shown in section 10.11.

$$a \leftarrow m \cdot a + \eta_t \nabla \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})\mathbf{W} \qquad\qquad \leftarrow \mathbf{W} - a$$

This allows getting over flatter surfaces of the optimization function quicker while also avoiding oscillations towards the end of the optimization process as we near the optimal point, see figure 22.

10.11 Weight-space symmetries

Multiple distinct weights compute the same prediction. I.e. given a network of the topology provided in figure 23, and we define the following variables:

$$v_i = \phi(\mathbf{w}_i, x)$$
$$f = w_1' v_1 + w_2' v_2$$
$$\mathbf{w} = [w_i, w_i']_i, \bar{\mathbf{w}} = [\bar{w}_i, \bar{w_i'}]_i$$

Suppose $\bar{\mathbf{w}}_2 = \mathbf{w}_1$, $\bar{\mathbf{w}}_1 = \mathbf{w}_2$, $\bar{\mathbf{w}_1'} = \mathbf{w}_2'$, $\bar{\mathbf{w}_2'} = \mathbf{w}_1'$, we then have:

$$f(\mathbf{x}; \bar{\mathbf{w}}) = f(\mathbf{x}; \mathbf{w})$$

which means that multiple local minima can be equivalent in terms of input-output mapping.

For instance, this is valid for the tanh activation function, where $\varphi(z) = -\varphi(-z)$

10.12 Avoiding overfitting

Due to the fact that neural networks have multiple parameters, they have the potential of overfitting the data. There are several countermeasures that can be adopted to mitigate this phenomenon:

- Early stopping: it consists in stopping SGD until it converges. This is done by monitoring the prediction performance on a validation set and stop trainign once the validation error starts to increase.
- Regularization: add penalty term to keep the weights small.

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{x}_i, y_i) + \lambda \|W\|_F^2$$

- Dropout: the key idea here is to randomly ignore hidden units during each iteration of SGD with probability p. At test time, we then multiply the weights by the probability p.
- Batch normalization: inputs are shifted and scaled through each layer. Batch normalization is a widely used technique that normalizes inputs to each layer according to mini-batch statistics.
  - It reduces internal covariate shift
  - Enables larger learning rates
  - It helps with regularization.

The algorithm for batch normalization goes as follows:

1. **Input**: values of x over a mini-batch: $\mathcal{B} = \{x_1, \dots, m\}$; Parameters to be learned: γ, β
2. **Output**: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\begin{aligned}
\mu_{\mathcal{B}} &\leftarrow \frac{1}{n}\sum_{i=1}^{m} x_i && \text{mini-batch mean} \\
\sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{n}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 && \text{mini-batch variance} \\
\hat{x_i} &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} && \text{normalize} \\
y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) && \text{scale and shift}
\end{aligned}$$

After this procedure, $\phi(\mathbf{wx}) \to \phi(\mathbf{w}\mathrm{B}_{\gamma,\beta}(\mathbf{x}))$