Géron, A.: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Tuesday, January 12, 2021 3:39 PM

Text book used for ECE612 Applied Machine Learning

Aurélien Géron -...

Part I: The Fundamentals of Machine Learning

Chapter 1: The Machine Learning Landscape

- Spam filters are where ML started off
- Machine learning is good for
    - Problems with lots of rules
    - Problems that need a solution found
    - Constantly fluctuating problems
    - Problems without enough insight
- Examples of ML
    - Image classification
    - Semantic segmentation
    - Natural Language Processing (NLP)
        - Text Classification
        - Text Summarization
        - Natural Language Understanding (NLU)
    - Regression
    - Speech Recognition
    - Anomaly Detection
    - Clustering
    - Data Visualization
        - Dimensionality Reduction
    - Recommender System
    - Reinforcement Learning
        - For game AI
- Types of ML Systems
    - Degree of human supervision (supervised, unsupervised, semisupervised, reinforcement learning)
        - Supervised Algorithms
            - k-Nearest Neighbors
            - Linear/Logistic Regression
            - Support Vector Machines (SVMs)
            - Decision Trees and Random Forests
            - Neural Networks
        - Unsupervised Algorithms
            - Clustering
                - K-Means
                - DBSCAN

                - Hierarchical Cluster Analysis (HCA)
            - Anomaly detection and novelty detection
                - One-class SVM
                - Isolation Forest
            - Visualization and dimensionality reduction
                - Principal Component Analysis (PCA)
                - Kernel PCA
                - Locally Linear Embedding (LLE)
                - t-Distributed Stochastic Neighbor Embedding (t-SNE)
            - Association Rule Learning
                - Apriori
                - Eclat
        - Semisupervised Techniques
            - Deep Belief Networks (DBNs)
                - Containing stacked Restricted Boltzmann Machines (RBMs)
        - Reinforcement Training (risk, reward, and penalty)
    - Learn in real time (online or batch learning)
        - Batch/Offline Learning
        - Online Learning
            - Can be done in mini-Batches
            - Out-of-core learning/Incremental Learning
                - Train on huge datasets in pieces until the whole dataset has been trained upon
    - Compare to known data or find patterns in new data (instance-based or model-based learning)
        - Instance-based learning
            - Determine measure of similarity between inputs
        - Model-based learning
            - Prediction based off of models
            - Requires model selection
                - Determine model accuracy based on a utility/fitness function (how well) or a cost function (how not-well)
- ML Challenges
    - Not enough data (curse of dimensionality!)
    - Bad (non-representative) data
        - Too much noise?
        - How much sampling bias?
        - Errors or outliers?
        - Irrelevant data or features?
    - Over/under-fitting
        - Overfitting may be fixed by model simplification, more training data, noise reduction
        - Underfitting may be fixed by model complexification, feature engineering, reduced regularization hyperparameters
- Testing and Validating ML Algorithms/Models
    - Test the model against itself
        - Testing and training sets
    - Model selection and hyperparameter tuning

Chapter 2: End-to-End Machine Learning Project

- Please refer to whatever files I create for this project in my school folder
- Performance Measures
    - Root Mean Square Error

- $RMSE(\boldsymbol{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}^{(i)}) - y^{(i)})}$
  - Where m is the number of instances used from the dataset
  - $x^{(i)}$ is the vector of feature values with their labels $y^{(i)}$
  - h is the system prediction function (AKA *hypothesis)*
- $Euclidean\ norm = \sqrt{\text{sum of squares } RMSE(\boldsymbol{X}, h)}$

○ Mean Absolute Error

- $MAE(\boldsymbol{X}, h) = \frac{1}{m}\sum_{i=1}^{m}|h(\boldsymbol{x}^{(i)}) - y^{(i)}|$
- $Manhattan\ norm = \text{sum of absolutes (MAE)}$

- A note on *virtualenv*:
  - To use, type "virtualenv *name of environment*" and it will create new python executables and libraries in their own folder

Chapter 3: Classification

- MNIST: Digits data set
  - Sci-kit learn dictionaries have:
    - DESCR key with a description of the data
    - data key with an array each instance in a row and each feature in a column
    - target key with labels
  - SGD (Stochastic Gradient Descent) for binary classification
    - Online learning advantage: one instance at a time
  - Methods of performance measuring
    - Cross-Validation
    - Confusion matrix
      - Measures of False/True Positives and False/True Negatives
    - Precision/Recall/Harmonic Mean
      - $Precision = \frac{TP}{TP + FP}$
      - $Recall = \frac{TP}{TP + FN}$
      - $Harmonic\ Mean\ (F_1) = \frac{TP}{TP + \frac{FN + FP}{2}}$
      - Receiver Operating Characteristics (ROC)
        - True Positive Rate (recall) vs False Positive Rate
        - Area under curve scores too
  - How about multi-classification?
    - One-vs-All or One-vs-One
      - OvO is best for large datasets
  - Multilabel classification might be good for recognizing individual people by face

Chapter 4: Training Models

- Training a linear regression model
  - "Closed-form"
    - Simply fit model as a weighted sum of inputs
      - $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$
        - $\hat{y}$ is the predicted value
        - n is the feature count
        - x_i is the i^{th} feature value
        - θ_j is the j^{th} model parameter
          - θ_0 is the bias term
          - $\theta_1, \theta_2, \ldots, \theta_n$ are feature weights

- Also represented as $\hat{y} = h_\theta(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$
- Now we need a way to train it: by using a performance measure
  - Find $\boldsymbol{\theta}$ that minimizes RMSE or MSE
  - $\text{MSE}(\boldsymbol{\theta}) = \text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)})^2$

- Minimize the performance function by using the *Normal Equation*
  - $\widehat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$
    - $\mathbf{y}$ is the vector of target values
    - Note that inversion of matrices can be computationally expensive
  - The pseudoinverse way
    - $\widehat{\boldsymbol{\theta}} = \mathbf{X}^+\mathbf{y}$
  - Singular Value Decomposition (SVD)
    - $\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathbf{T}$

      The pseudoinverse itself is computed using a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U}\,\boldsymbol{\Sigma}\,\mathbf{V}^\intercal$ (see `numpy.linalg.svd()`). The pseudoinverse is computed as $\mathbf{X}^+ = \mathbf{V}\boldsymbol{\Sigma}^+\mathbf{U}^\intercal$. To compute the matrix $\boldsymbol{\Sigma}^+$, the algorithm takes $\boldsymbol{\Sigma}$ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix. This approach is more efficient than computing the Normal Equation, plus it handles edge cases nicely: indeed, the Normal Equation may not work if the matrix $\mathbf{X}^\intercal\mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined.

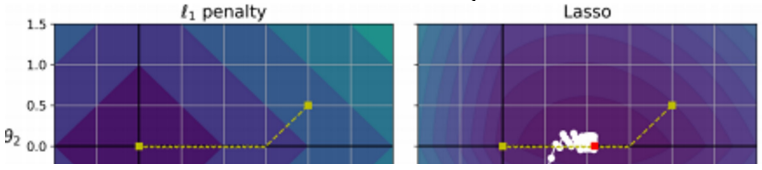
- Iterative optimization (Gradient Descent)
  - Adjustable starting point, learning rate, and whatnot
  - Problems with local minima (but great to use with MSE)
  - Try to scale features to be of equal size in order to converge faster (***Scikit StandardScaler***)
  - Batch Gradient Descent
    - Compute partial derivative of cost function wrt all parameters
      - $\frac{\partial}{\partial\theta_j}\text{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)})x_j^{(i)}$
      - $\nabla_{\boldsymbol{\theta}}\text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial\theta_1}\text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$
    - Slow since it uses the entire training set to compute things
    - Iterate this:
      - $\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\text{MSE}(\boldsymbol{\theta})$
    - Consider iterating with different learning rates and a check for when the relative change becomes less than some *tolerance* ϵ
  - Stochastic Gradient Descent
    - Take steps down the gradient by picking a random training feature each step
    - Constant bouncing could be bad, but might be good (bounce out of local minima)
    - Could lower the learning rate over time to decrease bouncing toward the end
      - Add a *learning schedule*
  - Mini-batch Gradient Descent
    - Each step takes a random small group of features (combine batch and stochastic)

        - Really good for hardware optimization (GPU)

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

- Polynomial regression
    - Ooh complexity
    - Watch out for overfitting with Learning Curves
    - Regularized linear models
        - Ridge Regression (Tikhonov Regularization)
            - Regularization term added to cost function : $\alpha \sum_{i=1}^{n} \theta_i^2$
                - Only during training
                - α defines how much regularization is to be done
            - Keeps weights as small as possible
            - $J(\mathbf{\theta}) = \text{MSE}(\mathbf{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$
                - Note: bias term is not regularized
            - Other representation
                - If $\mathbf{w}$ is the weight vector, then
                    - $\alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2 = \frac{1}{2} (\| \mathbf{w} \|_2)^2$
                    - $\| \mathbf{w} \|_2$ is the l_2 norm of the weight vector
                - To add to Gradient Descent, just add $\alpha \mathbf{w}$ to the MSE gradient vector
                    - $\hat{\mathbf{\theta}} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^T \mathbf{y}$
        - Lasso (Least Absolute Shrinkage and Selection Operator Regression) Regression
            - Uses the l_1 norm instead of half the square of the l_2 norm
                - $J(\mathbf{\theta}) = \text{MSE}(\mathbf{\theta}) + \alpha \sum_{i=1}^{n} |\theta_i|$
            - Usually eliminates weights of the least important features (automatically performs feature selection and outputs a *sparse* model)
            - Note that the Lasso cost function doesn't have a derivative when a parameter is equal to zero (except bias term)
                - Use the *subgradient vector* for gradient descent to keep things cool
                - $g(\mathbf{\theta}, J) = \nabla_{\mathbf{\theta}} \text{MSE}(\mathbf{\theta}) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix}$
                    - Where $\text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$



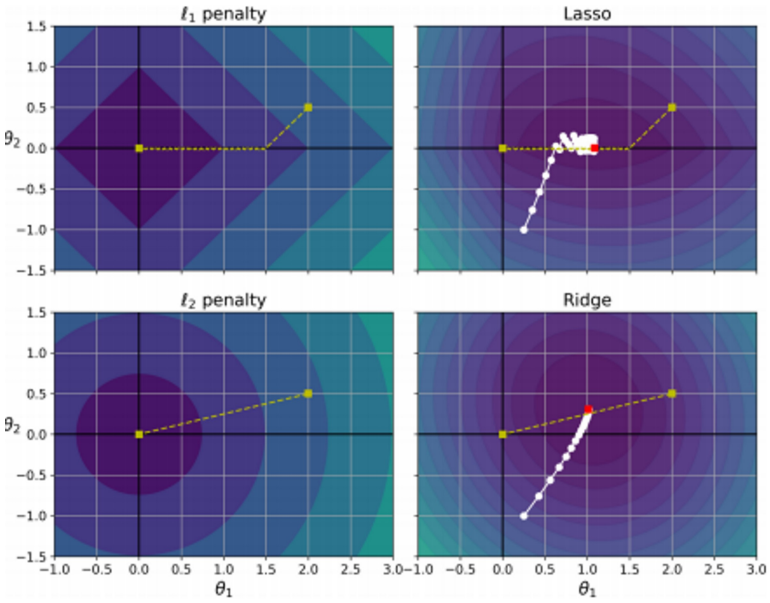
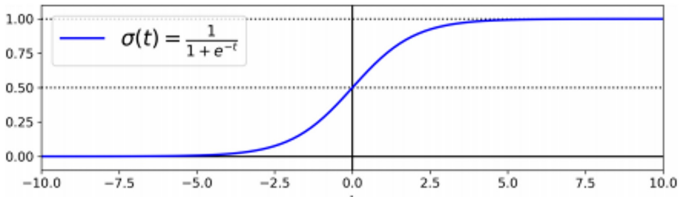


Figure 4-19. Lasso versus Ridge regularization

- Elastic Net (between Ridge and Lasso)
  - Mixing the regularization terms from both with a mixing ratio (r=0 is Ridge, r=1 is Lasso)
  - $J(\mathbf{\theta}) = \text{MSE}(\mathbf{\theta}) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$
- Early Stopping
  - Stop after validation error is at its minimum

- Logistic (Logit) regression
  - Probability that a particular feature belongs to a class (binary classification)
  - Like linear regression but instead outputs a logistic
    - $\hat{p} = h_{\mathbf{\theta}}(\mathbf{x}) = \sigma(\mathbf{x}^T \mathbf{\theta})$
      - Where $\sigma(\cdot) = \sigma(t) = \frac{1}{1+\exp(-t)}$
      - And t is called the *logit*
      - 

    - Then it makes a decision
      - $\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$
  - Training this model
    - Cost function
      - $c(\mathbf{\theta}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1-\hat{p}) & \text{if } y = 0 \end{cases}$
    - Log loss function
      - $J(\mathbf{\theta}) = -\frac{1}{m} \sum_{i=0}^{m} [y^{(i)} \log(\hat{p}^{(i)}) + (1-y^{(i)}) \log(1-\hat{p}^{(i)})]$
        - Uh, oh there's no closed-form solution for minimization!
        - But gradient descent totally works every time
      - $\frac{\partial}{\partial \theta_j} J(\mathbf{\theta}) = \frac{1}{m} \sum_{i=1}^{m} (\sigma(\mathbf{\theta}^T \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$

        - Can penalize Logistic regression with l_1 or l_2 norms (Scikit does second automatically)
- Softmax (normalized exponential) regression (Multinomial Logistic Regression)
    - Logistic with multiple classes
        - $s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$
            - Each class has a parameter vector usually stored as a parameter matrix $\boldsymbol{\Theta}$
        - The Softmax Function
            - $\hat{p}_k = \sigma(\mathbf{x})_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^{K} \exp(s_j(\mathbf{x}))}$
                - K is the number of classes
                - S(x) is a vector containing scores for each class for the instance x
                - $\sigma(\mathbf{x})_k$ estimated probability of x belonging to class k given the scores of each class for that instance
        - Prediction
            - $\hat{y} = \underset{k}{\operatorname{argmax}} \left(\sigma(\mathbf{x})_k\right) = \underset{k}{\operatorname{argmax}} s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \left(\boldsymbol{\theta}^{(k)T} \mathbf{x}\right)$
    - Only predict one class at a time (multi-class but single output)
    - Training
        - Cost function
            - $J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$
                - $y_k^{(i)}$ target probability that ith instance belongs to class k (usually 0 or 1)
            - $\boldsymbol{\theta}^{(k)} = \nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right) \mathbf{x}^{(i)}$
        - ASIDE: A note here on cross-entropy
            - $H(p, q) = -\sum_x p(x) \log q(x)$ for discrete distributions
            - Wrong assumptions about probabilistic bit representations lead to an cross-entropy amount greater than the ideal by an amount called *Kullback-Leibler (KL) divergence*

Chapter 5 Support Vector Machines (SVMs)

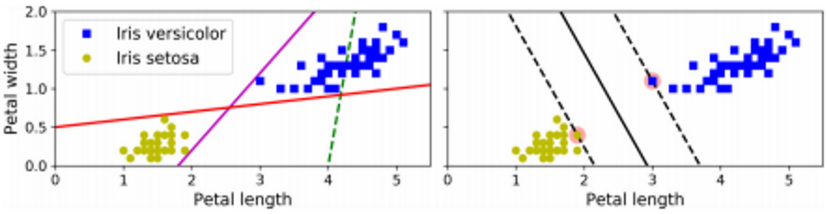
- Linear SVM Classification
    - Linearly separable classes separated by a boundary that maximizes the distance of that boundary from any member of either classes (large margin classification)
    - Decision boundary determined solely by the *support vectors* (the features at the 'edge' of the 'street') circled below
        - 

        Figure 5-1. Large margin classification
    - Soft Margin Classification
        - Opposite of Hard Margin (every instance must be off the 'street') where
        - Hyperparameter C which (when larger) reduces the street size and reduces generalizability

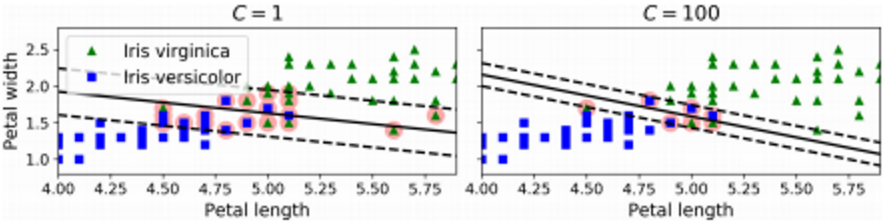
- 

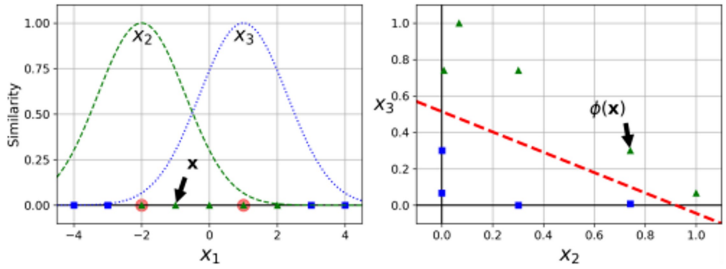
Figure 5-4. Large margin (left) versus fewer margin violations (right)
    - Scikit's LinearSVC class scales linearly, and does not permit the kernel trick (change precision with the epsilon/tolerance hyperparameter)
- Nonlinear SVM Classification
    - Perhaps add some more dimensions to your data and spline those lines
    - Use the *kernel trick*
    - May want to add features that reflect similarity between other features
        - Various Similarity Functions
            - Gaussian *Radial Basis Function (RBF)*
                - $\phi_\gamma(\mathbf{x}, l) = \exp(-\gamma \parallel \mathbf{x} - l \parallel^2)$
                - Choose value of $\gamma = 0.3$ (just cause)
                - Determine distance of each feature from some landmark feature
                - 

Figure 5-8. Similarity features using the Gaussian RBF
                - Using this as a kernel is good
            - Other kernels exist for specific purposes (like string kernels for words or DNA sequences)
    - *Table 5-1. Comparison of Scikit-Learn classes for SVM classification*

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

- SVM Regression (regression as well as classification)
    - Can also be used for outlier detection
- Under the Hood (How SVMs predict)
    - Notation: bias term is now *b*, feature weights are **w** and do not include bias
    - Decision Function and Predictions
        - SVM classifier computes this value, where positive is True class and negative is False class
            - $\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^T\mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^T\mathbf{x} + b \geq 0 \end{cases}$
            - Where $\mathbf{w}^T\mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$

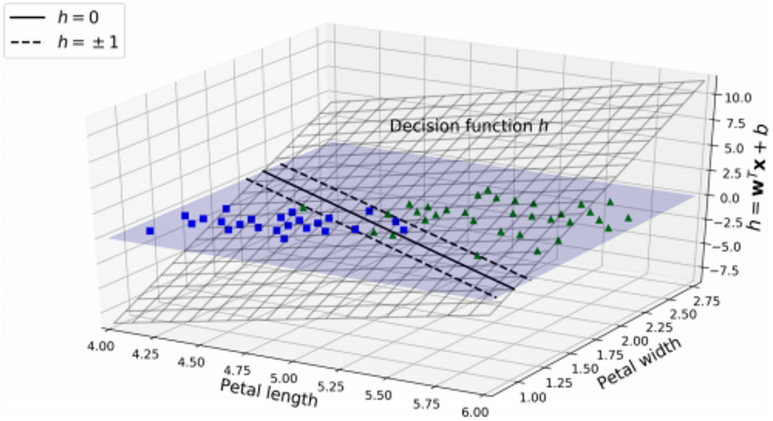
- 

Figure 5-12. Decision function for the iris dataset

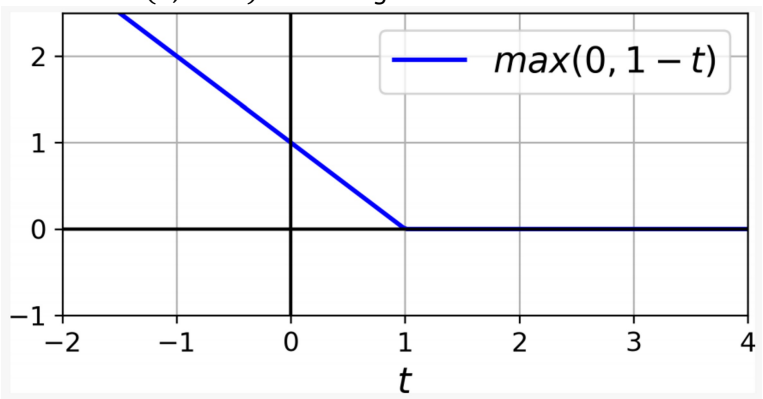
- Training Objective
  - Slope of decision function equal to norm of weight vector ($\| \mathbf{w} \|$)
    - Dividing this by a number greater than 1 will increase the 'size of the street'
  - Hard Margin Optimization
    - $\underset{\mathbf{w},b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} = \frac{1}{2}\| \mathbf{w} \|^2$
    - subject to $t^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m$
  - Soft Margin Optimization (with slack $\zeta \geq 0$)
    - Slack tells us how much each instance is allowed to violate the decided margin
    - $\underset{\mathbf{w},b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$
    - subject to $t^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0$ for $i = 1, 2, \dots, m$
- Quadratic Programming (QP) (that's training: convex quadratic optimization problems with linear constraints)
  - Many ways to solve!
  - Here's one!
    - $\underset{p}{\text{Minimize}} \quad \frac{1}{2}\mathbf{p}^{\mathrm{T}}\mathbf{H}\mathbf{p} + \mathbf{f}^{\mathbf{T}}\mathbf{p}$
    - subject to $\mathbf{A}\mathbf{p} \leq \mathbf{b}$
  - where $\begin{cases} \mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} & \text{is an } n_p \times n_p \text{ matrix}, \\ \mathbf{f} & \text{is an } n_p\text{-dimensional vector}, \\ \mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} & \text{is an } n_c\text{-dimensional vector}. \end{cases}$
    - Note that the expression $\mathbf{A}\,\mathbf{p} \leq \mathbf{b}$ defines n_c constraints: $\mathbf{p}^{\mathrm{T}}\,\mathbf{a}^{(i)} \leq b^{(i)}$ for $i = 1, 2, \cdots, n_c$, where $\mathbf{a}^{(i)}$ is the vector containing the elements of the i^{th} row of $\mathbf{A}$ and $b^{(i)}$ is the i^{th} element of $\mathbf{b}$.
  - Hard margin with following parameters
    - $n_p = n + 1$, where n is the number of features (the +1 is for the bias term).
    - $n_c = m$, where m is the number of training instances.
    - $\mathbf{H}$ is the $n_p \times n_p$ identity matrix, except with a zero in the top-left cell (to ignore the bias term).
    - $\mathbf{f} = 0$, an n_p-dimensional vector full of 0s.
    - $\mathbf{b} = -1$, an n_c-dimensional vector full of −1s.
    - $\mathbf{a}^{(i)} = -t^{(i)}\,\dot{\mathbf{x}}^{(i)}$, where $\dot{\mathbf{x}}^{(i)}$ is equal to $\mathbf{x}^{(i)}$ with an extra bias feature $\dot{\mathbf{x}}_0 = 1$.

- The Dual Problem (for kernel trick)
  - Primal problem and dual problem (SVMs have the same solution)
  - (Dual Problem of the linear SVM objective below derived from Primal Problem in Appendix C)
    - $\underset{\alpha}{\text{minimize}} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)\mathbf{T}}\mathbf{x}^{(j)} - \sum_{i=1}^{m}\alpha^{(i)}$
    - subject to $\alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \ldots, m$
  - Find $\hat{\alpha}$ that minimizes the above equation, then compute $\hat{\mathbf{w}}$ and $\hat{b}$ that solve the primal problem
    - $\hat{\mathbf{w}} = \sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\mathbf{x}^{(i)}$
    - $\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} t^{(i)} - \hat{\mathbf{w}}^{\mathrm{T}}\mathbf{x}^{(i)}$
- Kernelized SVMs (THE kernel trick)
  - Map to a higher dimension (mind = blown) using a mapping function phi
    - $\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$
  - Example: apply this to 2d vectors a and b
    - $\phi(\mathbf{a})^{\mathbf{T}}\phi(\mathbf{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2$

      $= (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1^{\mathrm{T}} \\ a_2 \end{pmatrix}\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 = (\mathbf{a}^{\mathrm{T}}\mathbf{b})^2$
    - Don't necessarily have to transform every training feature, because of this dot product thingy
  - Kernel: "A function capable of computing the dot product… based only on the original vectors… without having to compute (or even know about) the transformation {phi}" [pg 232]
  - Common Kernels
    - $\begin{aligned} \text{Linear:} \quad & K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^{\mathrm{T}}\mathbf{b} \\ \text{Polynomial:} \quad & K(\mathbf{a}, \mathbf{b}) = (\gamma\mathbf{a}^{\mathrm{T}}\mathbf{b} + r)^d \\ \text{Gaussian RBF:} \quad & K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma\|\mathbf{a} - \mathbf{b}\|^2\right) \\ \text{Sigmoid:} \quad & K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma\mathbf{a}^{\mathrm{T}}\mathbf{b} + r\right) \end{aligned}$
    - Mercer's Thm states that any function can be a kernel mapping to some high dimensional space if it follows "Mercer's conditions"
      - Some don't have to satisfy all of them (Sigmoid)
  - Now we plug the formula for the $\hat{\mathbf{w}}$ into the decision function for a particular feature instance (what comes out is dot products so it's easy)
    - $h_{\hat{\mathbf{w}},\hat{b}}\left(\phi(\mathbf{x}^{(n)})\right) = \hat{\mathbf{w}}^{\mathrm{T}}\phi(\mathbf{x}^{(n)}) + \hat{b} = \left(\sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\phi(\mathbf{x}^{(i)})\right)^{\mathrm{T}}\phi(\mathbf{x}^{(n)}) + \hat{b}$

      $= \sum_{\mathrm{i}=1}^{\mathrm{m}}\hat{\alpha}^{(i)}t^{(i)}\left(\phi(\mathbf{x}^{(i)})^{\mathrm{T}}\phi(\mathbf{x}^{(n)})\right) + \hat{b}$

      $= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\hat{\alpha}^{(i)}t^{(i)}K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b}$

- $\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \hat{\mathbf{w}}^{\mathrm{T}}\phi(\mathbf{x}^{(i)})\right) = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \left(\sum_{j=1}^{m}\hat{\alpha}^{(j)}t^{(j)}\phi(\mathbf{x}^{(j)})\right)^{\mathrm{T}}\phi(\mathbf{x}^{(i)})\right)$
  $= \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \sum_{\substack{i=1 \\ \hat{\alpha}^{(j)}>0}}^{m}\hat{\alpha}^{(j)}t^{(j)}K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})\right)$
- Online SVMs
  - Gradient Descent
    - $J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{i=1}^{m}\max\left(0, 1 - t^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b)\right)$
      - Where $\max(0, 1-t)$ is the *hing loss* function
      - 

Chapter 6: Decision Trees

- *Gini* score measures impurity (how often the training samples the node applies to belong in the same class together)
  - $G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$
  - Better for applications with a 'dominant class'
    - Where $p_{i,k}$ is the ratio of class *k* instances among the training instances in the i^{th} node.
- Decision trees are often called white box models since their calculations are easy to replicate and intuitive
- *Classification and Regression Tree (CART)* Algorithm
  - Progressively split the set further and further using single feature *k* and threshold t_k with parameters that produce the "purest" separations
  - Cost Function
    - $J(k, t_k) = \frac{m_{left}}{m}G_{left} + \frac{m_{right}}{m}G_{right}$
    - where $\begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$
  - Recursion until *max_depth* is reached or if any split would not improve impurity
    - NOTE: This is a greedy algorithm, and as such finding an optimal splitting from the perspective all levels down is NP-Complete with O(exp(m)) (super impractical)
    - Predictions are fast tho $O(\log_2(m))$
    - Training is $O(n \times m\log_2(m))$
  - We can also use Entropy as the cost funciton
    - Shannon Entropy: $H_i = -\sum_{k=1}^{n} p_{i,k}\log_2(p_{i,k})$
    - Best for applications requiring balanced trees

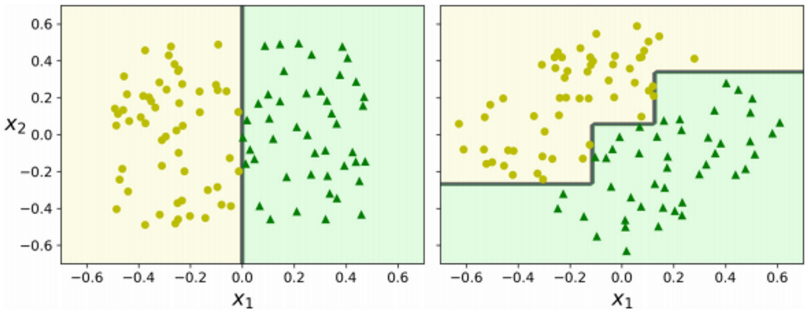
- Decision trees are nonparametric, so they risk overfitting, try using these parameters to restrict the degrees of freedom of the learning
    - Max_depth: self explanatory
    - Min_samples_split: the minimum number of samples a node must have before it can split
    - Min_samples_leaf: the minimum number of samples a leaf node must have
    - Min_weight_fraction_leaf: same as min_samples_leaf but expressed as a fraction of the total number of weighted instances
    - Max_features: the maximum number of features that are evaluated for splitting at each node
- We can do regression too
    - Now we split with a cost function dependent upon MSE
        - $J(k, t_k) = \frac{m_{left}}{m}\text{MSE}_{left} + \frac{m_{right}}{m}\text{MSE}_{right}$
        - where $\begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} . \end{cases}$
- Decision trees Limitations
    - All decision boundaries are axis-orthogonal, so results are subject to rotation challenges
        - 

        Figure 6-7. Sensitivity to training set rotation
    - Sometimes better to PCA things to orient the data
    - Models are Stochastic, so various decision boundaries may develop for a particular training set

Chapter 7 Ensemble Learning and Random Forests

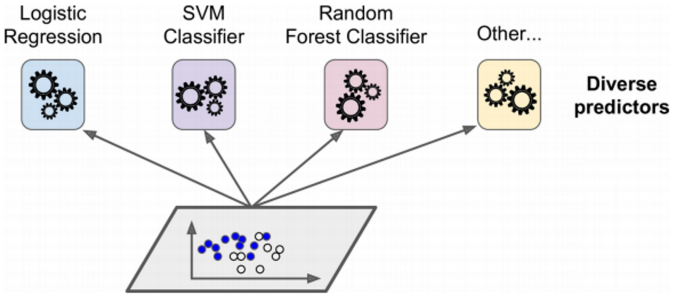
- The wisdom of the crowd, that's the idea here
- Voting Classifiers
    - Take all them classifiers and have 'em vote
        - 

        Figure 7-1. Training diverse classifiers
    - My Opinion: Some classifiers have innate better predictions towards some problems so by having a diverse background of classifiers working on the problem, the outcome often can come out on top.
    - How it works for realsies: The Law of Large Numbers
        - Gotta watch out about dependence though. This only applies for independent samples and ensemble learning may not be so independent since they are trained on the same data

    - Hard vs Soft voting: Soft voting gives weight to the classifier that is the most confident about its prediction whereas hard voting gives the same weight to everyone
- Bagging and Pasting
    - Using very different training algorithms for your ensemble predictors, or use different random subsets of the data
    - Bagging: Sampling *with* replacement from training set (aka *bootstrap aggregating*)
        - Higher resulting bias, but lower variance
    - Pasting: Sampling *without* replacement
    - Then aggregate results
        - For classification, use hard voting usually
        - For regression, take the average of everyone
- Out-of-Bag Evaluation
    - When bootstrapping happens, a random subset is used to train, but what about those that aren't picked? Those are the *out-of-bag (oob)* instances
        - We can validate on these instances
        - Set *oob_score=True* with the BaggingClassifier in SciKit to do it automatically
- Random Patches and Random Subspaces
    - Controlled via 2 hyperparameters
        - Max_features: self-explanatory
        - Bootstrap_features:
        - Same as max_samples and bootstrap, but instead of doing it for instances it's for features
    - Nice for high-dimensional inputs (i.e. images?)
    - Random Patches: randomly sampling instances *and* features
    - Random Subspaces: keep all training instances, but randomly sample features
- Random Forests
    - Extra-Trees (Extremely Randomized Trees)
        - Even more randomness can be introduced by setting random thresholds for features instead of searching for the best threshold (how decision trees usually work)
        - More bias, lower variance
        - Runs real fast though because there is no computation needed for decision boundary determination
    - Feature Importance
        - Generally, the more a feature influences the impurity of a node, the more important that node is
        - This is automatically done by SciKit's random forests
- Boosting (Hypothesis Boosting)
    - Anytime you can combine a bunch of weak learners to become one big strong learner. Usually learners are trained sequentially with each subsequent learner being influenced by the errors of the last.
    - AdaBoost
        - The next predictor takes a closer look at the instances/features that the last learner didn't do well on
        - Results in harder and harder cases to learn on

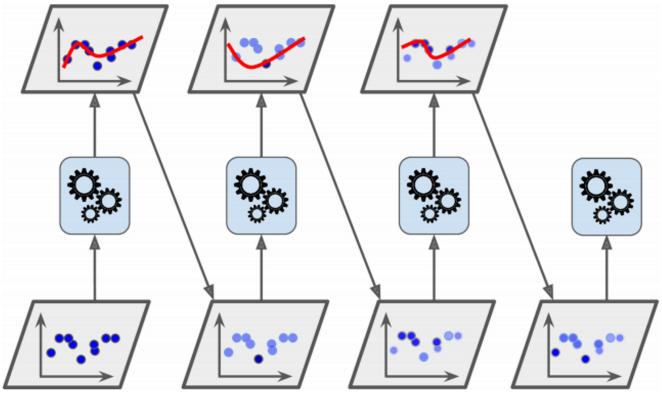
- 

  Figure 7-7. AdaBoost sequential training with instance weight updates

  - Algorithm
    - Initialize
      - All weights set equal: $w^{(i)} = \frac{1}{m}$
        - Where m is the number of training instances
    - Compute error rates

      Equation 7-1. Weighted error rate of the j^{th} predictor

      - $$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$ where $\hat{y}_j^{(i)}$ is the j^{th} predictor's prediction for the i^{th} instance.
    - Update weights accordingly
      - Update weight of predictor
        - $\alpha_j = \eta \log \frac{1 - r_j}{r_j}$
      - Update instance weights

        for $i = 1, 2, \ldots, m$
        - $$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$
      - Normalize weights
        - Divide by $\sum_{i=1}^{m} w^{(i)}$
    - Stop after perfect predictor found or when desired number of predictors is produced
    - Predict
      - $\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^{N} \alpha_j$ where N is the number of predictors
  - SciKit uses a multiclass AdaBoost algorithm *Stagewise Additive Modeling using a Multiclass Exponential loss function (SAMME)*
    - Or *SAMME.R* (R for "real") relying on class probabilities rather than predictions
  - Note: Decision Stump is a depth=1 decision tree
- Gradient Boosting
  - Instead of tweaking rates, the next predictor in the line is made to fit any *residual error* from the last predictor
  - XGBoost is an optimized version of Gboost

- Stacking (*stacked generalization*)
  - Train a model to determine how voting should be done (not hard, not soft, but somewhere in between!)

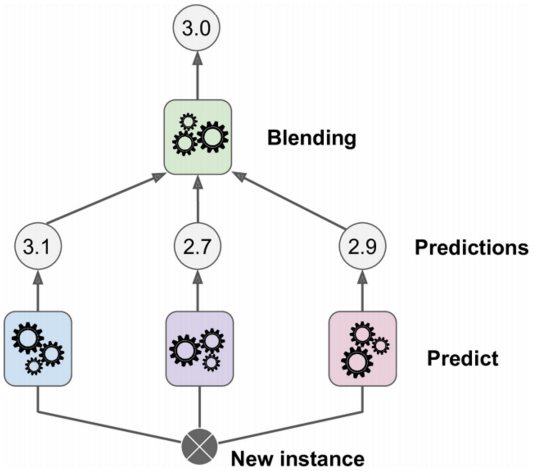
- 

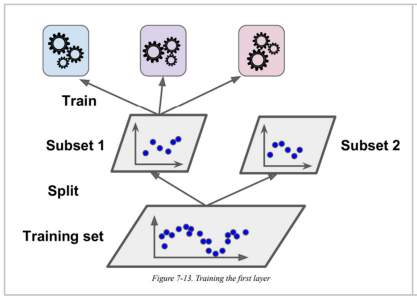
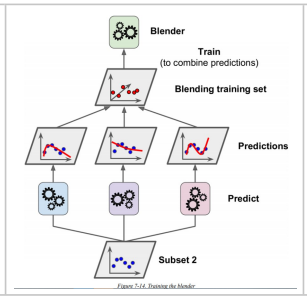
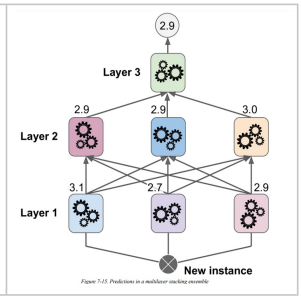
Figure 7-12. Aggregating predictions using a blending predictor
  - How do we do it?
    - Use a *hold-out set*
      1. Split training into 2 subsets
         a) 1st subset trains first layer predictors
            i) 1st layer predicts on the subset left for the 2nd layer
            ii) Output 3 predicted values to use as input for the blender
         b) Then proceed to next layer
      2. This was a rather confusing and not well explained topic, but it's cool, here're some pictures
    - 

Figure 7-13. Training the first layer




    - Scikit does not have an implementation (as of this book's writing
      - Other library open source (*brew*)

Chapter 8: Dimensionality Reduction

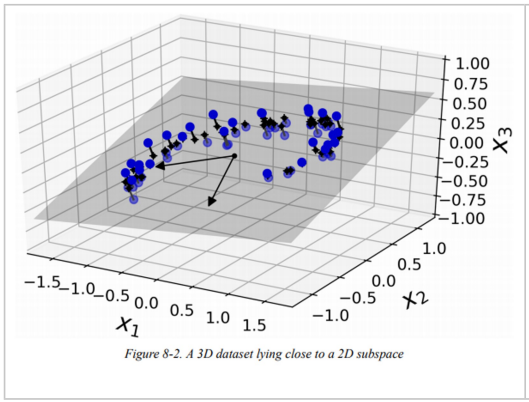
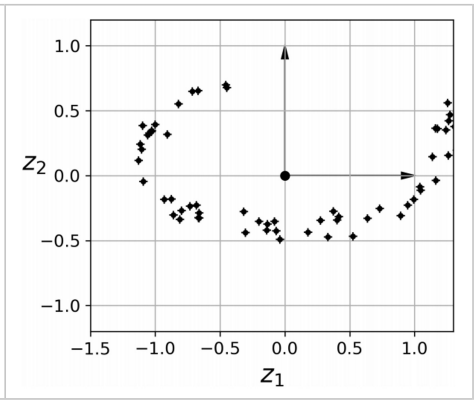
- The Curse of Dimensionality
  - Sparsity exists greatly in higher-dimensional spaces
  - An insane amount of training data would be necessary for good predictability in these spaces
- Main Approaches for Dimensionality Reduction
  - Projection (self-explanatory)
    - 

Figure 8-2. A 3D dataset lying close to a 2D subspace


  - Manifold Learning
    - What's a manifold? Well it's a 2D (or lower dimensional dataset) that can be turned

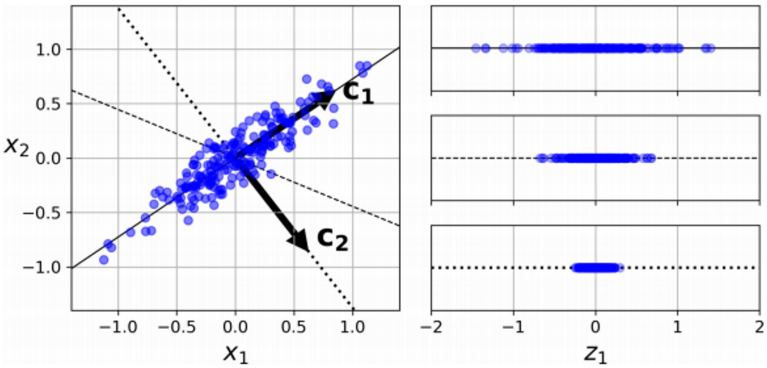
and twisted in higher-dimensional ways (or another higher-dimensional dataset)
    - *Manifold Hypothesis*: Most real-world datasets existing in a higher dimension actually lie close to some lower-dimensional manifold
  - Learn by modeling the dataset as if it were a lower dimensional manifold
- Principal Component Analysis (PCA)
  - Identifying the hyperplane closest to the data, then projects the data onto it. Uses variance to determine which plane is closes
  - 

    Figure 8-7. Selecting the subspace to project on
    - Choose the plane that when projected upon, yields the highest variance
      - Or collection of planes in lower dimensions (the principal components)
  - Find by using Singular Value Decomposition
    - $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$
      - V often contains the unit vectors defining the principal components
        - $\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$
    - Note: center your data first
  - Projecting Down to *d* Dimensions
    - Project into hyperplane with first *d* components
      - $\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$
      - $\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}}\mathbf{W}_d^T$
  - Explained variance ratio
    - Each component has this and it shows what portion of the dataset's variance is represented by that principal component
  - Choosing the Right Number of Dimensions
    - Choose the number of Dims that add up to a sufficiently large portion of the variance (96%?)
  - PCA for Compression
    - You can compress to a certain number of dimensions and recover from it
  - Randomized PCA
    - Randomly reconstructs the data according to the PCs
  - Incremental PCA
    - Uses pieces of the data instead of the whole thing
- Kernel PCA
  - Think kernel trick, with PCA. Yea, that

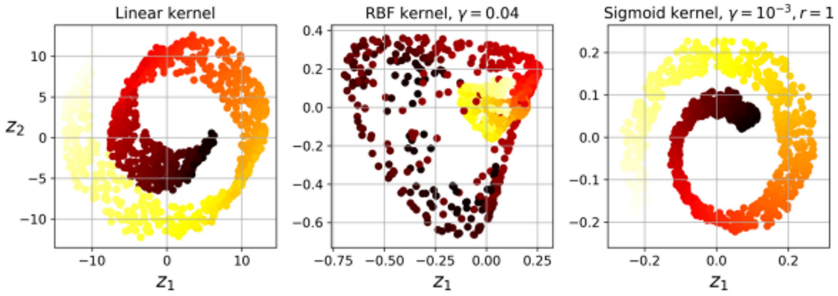
- 

Figure 8-10. Swiss roll reduced to 2D using kPCA with various kernels

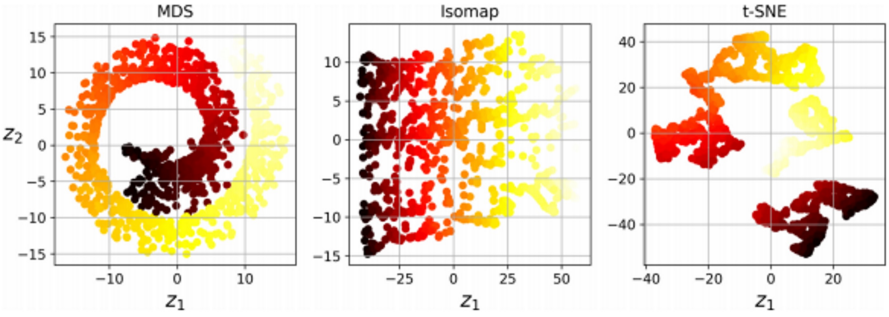
    - Selecting a Kernel and Tuning Hyperparameters
        - SciKit can do this
- Locally Linear Embedding (LLE)
    - Uses similarity metrics
    - How do?
        - Find the nearest neighbors of each training instance and reconstruct it uses these instances
        - $$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \mathbf{x}^{(j)} \right)^2$$
        - $$\text{subject to } \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}$$
        - Then reduce dimensionality
        - $$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$
- Other reduction techniques include:
    - Random projections
    - Multidimensional Scaling (MDS)
    - Isomap
    - t-Distributed Stochastic Neighbor Embedding (t-SNE)
    - Linear Discriminant Analysis (LDA)
    - 

Figure 8-13. Using various techniques to reduce the Swill roll to 2D

Chapter 9: Unsupervised Learning Techniques

- Clustering
    - K-Means
        - Can be a hard classifier or a soft one
        - [Cam, you already know this one]
        - How do?
            - Centroids are placed down randomly
                - Or choose em yourself
                - Or run a few times and pick the best (lowest model inertia)

                - Or use K-Means++
                    - Choose one centroid from the training set
                    - New centroid with a new instance with probability $\frac{D(\vec{i})}{\sum_{j=1}^{m} D(\vec{j})}$
                        - Farther centroids more likely to be picked than closer ones
                    - Repeat for all necessary centroids
            - They are 'pulled' towards the center of clusters
        - Other improvements
            - Accelerated and mini-batch
                - Accelerated kept track of distances between points and uses the triangle inequality to skip unnecessary distance measures
                - Watch out for the lower performance of batch learning here
        - Finding optimal number of clusters varies
            - Highest Silhouette score
            - Inertia elbow
            - Gap statistic (learned from Cohen)
    - Clustering for image segmentation is huge
    - Clustering can also be used for preprocessing
    - Can also be used with Semi-Supervised learning
    - DBSCAN
        - Clustering by areas of high density
        - And even HDBSCAN (hierarchical)
    - Other Algorithms
        - Agglomerative Clustering
        - BIRCH
        - Mean-Shift
        - Affinity Propagation
        - Spectral Clustering
- Gaussian Mixtures Model (GMM)
    - Assume a bunch of underlying Gaussian probability models
    -

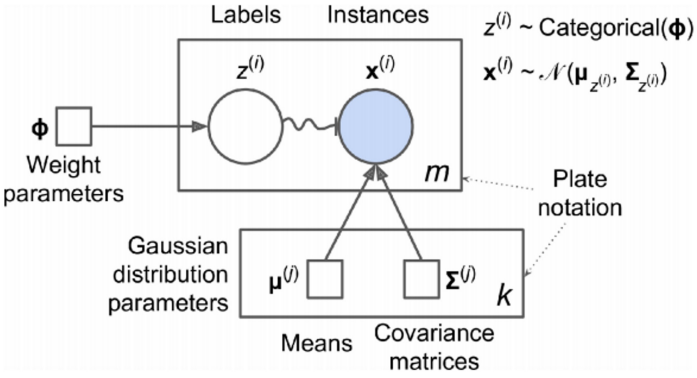
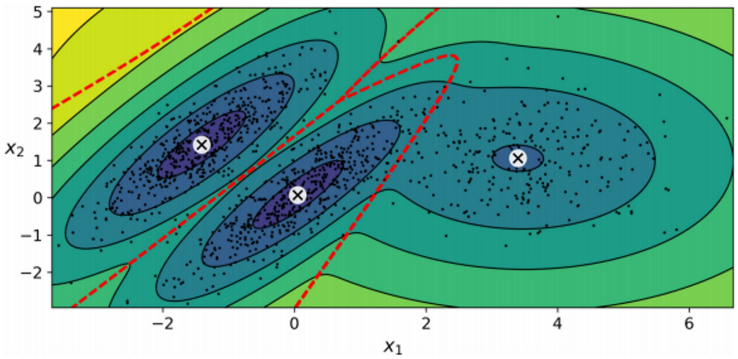


Figure 9-16. A graphical representation of a Gaussian mixture model, including its parameters (squares), random variables (circles), and their conditional dependencies (solid arrows)

- The circles represent random variables.
- The squares represent fixed values (i.e., parameters of the model).
- The large rectangles are called *plates*. They indicate that their content is repeated several times.
- The number at the bottom right of each plate indicates how many times its content is repeated. So, there are m random variables $z^{(i)}$ (from $z^{(1)}$ to $z^{(m)}$) and m random variables $\mathbf{x}^{(i)}$. There are also k means $\boldsymbol{\mu}^{(j)}$ and k covariance matrices $\boldsymbol{\Sigma}^{(j)}$. Lastly, there is just one weight vector $\boldsymbol{\phi}$ (containing all the weights $\phi^{(1)}$ to $\phi^{(k)}$).
- Each variable $z^{(i)}$ is drawn from the *categorical distribution* with weights $\boldsymbol{\phi}$. Each variable $\mathbf{x}^{(i)}$ is drawn from the normal distribution, with the mean and covariance matrix defined by its cluster $z^{(i)}$.
- The solid arrows represent conditional dependencies. For example, the probability distribution for each random variable $z^{(i)}$ depends on the weight vector $\boldsymbol{\phi}$. Note that when an arrow crosses a plate boundary, it means that it applies to all the repetitions of that plate. For example, the weight vector $\boldsymbol{\phi}$ conditions the probability distributions of all the random variables $\mathbf{x}^{(1)}$ to $\mathbf{x}^{(m)}$.
- The squiggly arrow from $z^{(i)}$ to $\mathbf{x}^{(i)}$ represents a switch: depending on the value of $z^{(i)}$, the instance $\mathbf{x}^{(i)}$ will be sampled from a different Gaussian distribution. For example, if $z^{(i)}=j$, then $\mathbf{x}^{(i)} \sim \mathcal{N}\left(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)}\right)$.
- Shaded nodes indicate that the value is known. So, in this case, only the random variables $\mathbf{x}^{(i)}$ have known values: they are called *observed variables*. The unknown random variables $z^{(i)}$ are called *latent variables*.

- Can even create new instances of the data using this model
- 

gure 9-17. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model

- Can modify what the underlying Gaussian distributions could look like by modifying the covariance parameter to either be spherical, diagonal, or tied
- And Bayesian models!

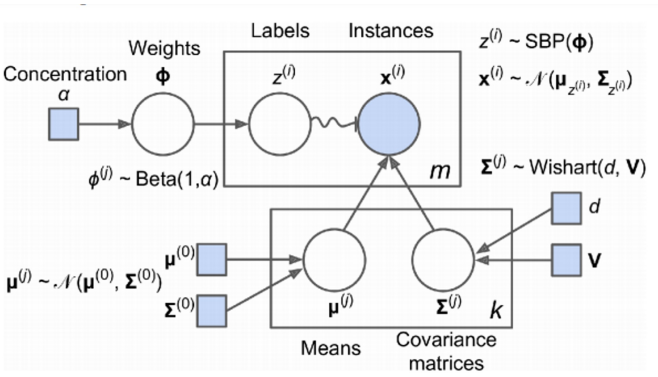
- 

  Figure 9-22. Bayesian Gaussian mixture model

- Other algorithms for anomaly and novelty detection
  - PCA
  - Fast-MCD (Minimum Covariance Determinant)
  - Isolation Forest
  - Local Outlier Factor (LOF)
  - One-class SVM

Part II Neural Networks and Deep Learning
Chapter 10 Introduction to Artificial Neural Networks with Keras

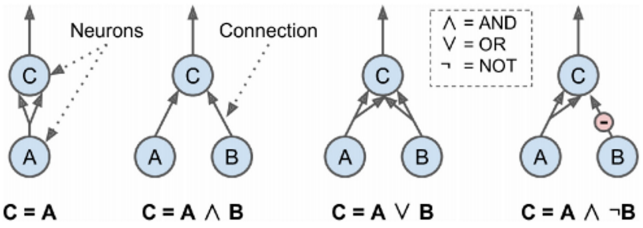
- First mention of neural networks: https://link.springer.com/article/10.1007%2FBF02478259
  - Spawned *connectionism* encompassing the field of studying neural networks
- Book goes over biological stuff.
- Simple perceptrons implementing logical functions
  - 

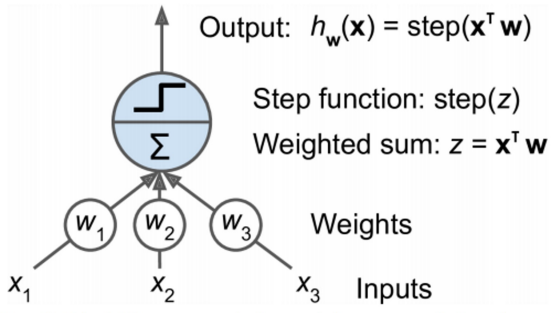
    Figure 10-3. ANNs performing simple logical computations
  - THE Perceptron OR Threshold Logic Unit (TLU) OR Linear Threshold Unit (LTU)
    - 

      Figure 10-4. Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function
  - Other step functions
    - $\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$
    - $\text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$
  - A sample perceptron architecture

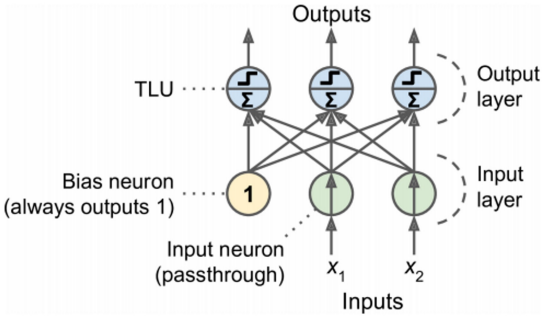
- 

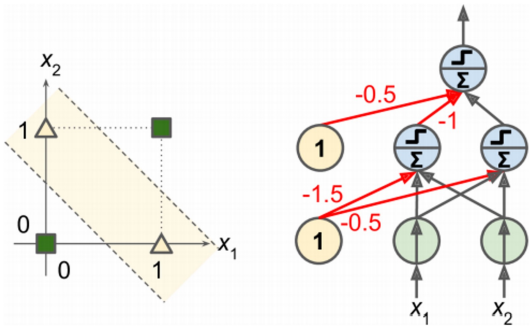
Figure 10-5. Architecture of a Perceptron with two input neurons, one bias neuron, and three output neurons
    - Outputs equation : $h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$
        - Where ϕ is whatever the *activation function* is
- Training occurs with *Hebb's Rule* that neurons that paths that are fired more often become stronger *(Hebbian Learning*)
    - Perceptron Learning rule (correcting for error in the output
        - $w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$
            - Where η is the learning rate
    - Normal perceptron decision boundaries are linear, but MLPs (multi-layer perceptrons) are different
        - MLPs add more lines and can solve the XOR problem usually impossible by simple perceptrons
        - 

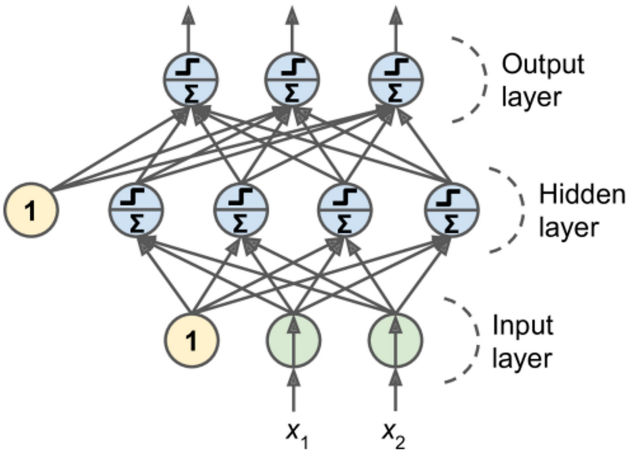
Figure 10-6. XOR classification problem and an MLP that solves it
        - General MLP architecture with one-hidden layer and bias neurons shown
        - 

Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)
            - When networks like this send info only forward, they are referred to as *feedforward neural networks (FNNs)*
        - Training MLPs (backpropagation paper: https://apps.dtic.mil/sti/pdfs/ADA164453.pdf)
            - One forward propagation and then one backward propagation trains a network with gradient descent
            - Automatic computing of gradients is done with various models of *automatic*

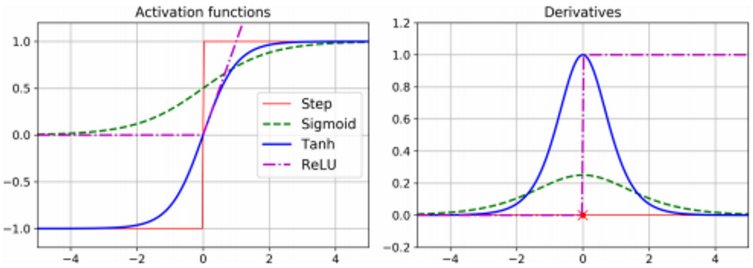
differentiation (autodiff) with backpropagation using the *reverse-mode autodiff* technique
    - Initialize all weights randomly
    - Operates in mini-batches and passes through the entire training set many times (each pass through the set is an *epoch*)
    - Pass the batch through from input -> output and record all neuron outputs
    - Determine the output error using some loss function
    - Using the chain rule, determine which connections contributed the most to this error from the previous later and carry back through to the input
    - Using the computed error gradients, perform a gradient descent and tweak weights as necessary
- For gradient descent to work, the activation functions must be *sigmoid functions* since the step functions have undefined derivatives or zeroed derivatives (when this is the case, GD can't make any progress)
    - Sigmoid Function: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Other functions work too
    - Hyperbolic Tangent: $\tanh(z) = 2\sigma(2z) - 1$
        - Output between -1 and 1 making centering easier, thereby speeding up training
    - Rectified Linear Unit: $\text{ReLU}(z) = \max(0, z)$
        - Very fast, but derivative in the negative zone is zero, and at zero it is undefined
        - Has no max output, which apparently helps GD
        - Smooth ReLU with *softplus*
            - $\text{softplus}(z) = \log(1 + \exp(z))$
    - Summary
        - 

Figure 10-8. Activation functions and their derivatives
        - Error functions include mean squared error (MSE), mean absolute error, or Huber loss (technically both)
        - Depending on your output neurons and activations, you can classify or regress
            - A *softmax layer* ensures that the outputs of the output layer all add up to one (showing probabilities for multiple exclusive classes)
- Keras
    - Needs a computation backend (TensorFlow, Microsoft Cognitive Toolkit (CNTK), or Theano)
        - Other backends include Apache MXNet, Apple's Core ML, JavaScript, TypeScript (Keras in browser), PlaidML (GPUs)
        - Book uses TensorFlow's Keras implementation aptly named *tf.keras*

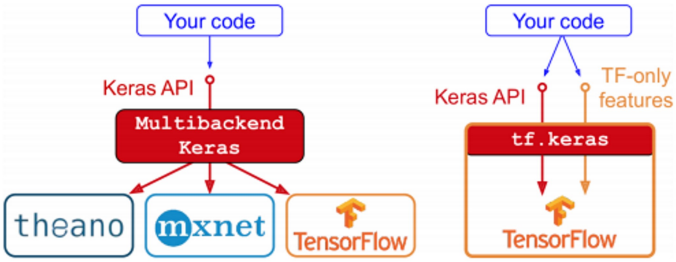
- 

Figure 10-10. Two implementations of the Keras API: multibackend Keras (left) and tf.keras (right)

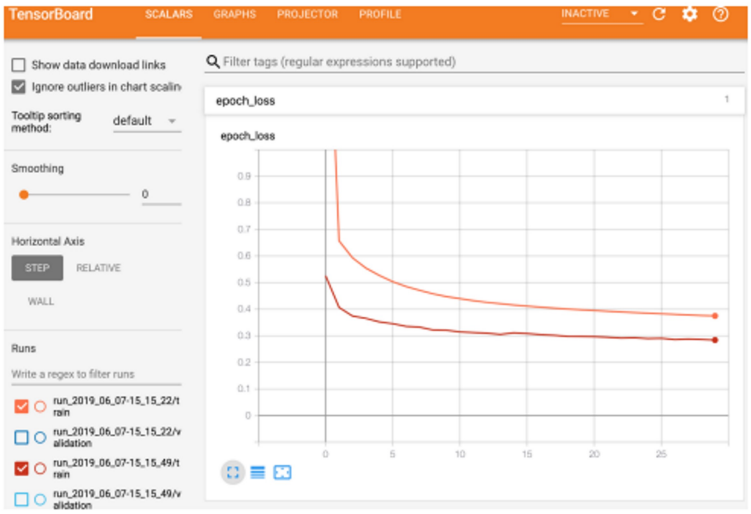
    - A close Keras substitute is PyTorch
        - Note: as of writing, Tensorflow is not available for Python 3.9, 3.8 works though
- Went through a bunch of Coding shtuffs
    - Models are saved as an .h5 file
        - Only for Sequential or Functional APIs. Weights will have to be saved and loaded for subclassing models
    - TensorBoard is good for visualization and it has a server backend for ease of browsing
        - ```
          $ tensorboard --logdir=./my_logs --port=6006
          TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
          ```
        - ```
          %load_ext tensorboard
          %tensorboard --logdir=./my_logs --port=6006
          ```
        - 

Figure 10-17. Visualizing learning curves with TensorBoard

    - Other python libraries for hyperparameters
        - Hyperopt
        - Hyperas, kopt, Talos
        - Keras Tuner
        - SciKit-Optimize (skopt)
        - Spearmint
        - Hyperband
        - Sklearn-Deap

Chapter 11: Training Deep Neural Networks

- A few problems you may run into
    - Vanishing/exploding gradients (especially in the deeper layers)
    - Not enough data for the size of the network
    - Slooooooooooooooooooooow training
    - Overfitting, my guy
- Vanishing/Exploding Gradients Problems
    - Vanishing
        - Gradients tend to get smaller the deeper into the network you go
        - Thus, during backpropagation, the lack of a gradient barely changes the weights of

the innermost neurons
    - Sometimes the exact opposite can happen (exploding)
- Paper sheds some light on why this problem exists: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
    - Seems that it was the sigmoid function that caused issues
    - Variance of each layer was increasing/decreasing relative to the inputs
        - So what if we ensure that the various remains the same after each layer (feedforward and during backpropagating)
    - *Xavier/Glorot initialization*
        - Can't guarantee variance requirement unless equal number of input neurons and output neurons (*fan-in* and *fan-out*)
        - So initialize the connection weights according to his formula, $fan_{\text{avg}} = \frac{(fan_{\text{in}}+fan_{\text{out}})}{2}$
        - It changes with various activation functions
            - Logistic: $\sigma^2 = \frac{1}{fan_{\text{avg}}}$
            - Or a uniform distribution between -r and +r: $r = \sqrt{\frac{3}{fan_{\text{avg}}}}$
    - *LeCun Initialization*
        - Replace in the previous equation fan_{avg} with fan_{in}.
        - Genevieve Orr and Klaus-Robert Müller recommended the above with $fan_{\text{in}} = fan_{\text{out}}$
        - This was a huge good thing and sped up deep learning training considerably
        - Other activations:
            - *Table 11-1. Initialization parameters for each type of activation function*

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{\text{avg}}$
He	ReLU and variants	$2 / fan_{\text{in}}$
LeCun	SELU	$1 / fan_{\text{in}}$

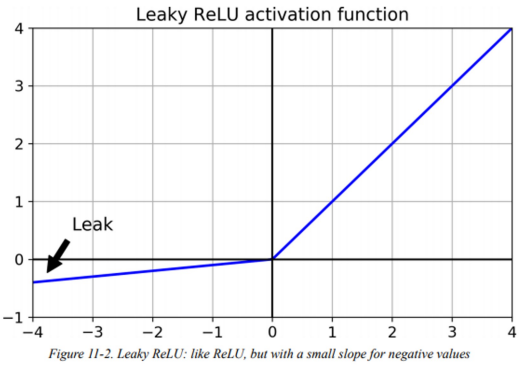
        - Modify initialization parameters with kernel_initializer=***
- Nonsaturating Activation functions
    - The activation function that is chosen is a very important point!
    - ReLU is really good for deep networks
        - Dying ReLU is a problem them, effectively killing neurons by training their outputs to only output zero always
            - Solve this with *leaky ReLU*
                - $\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z)$
                - 

Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values
                    - Where alpha defines the leak amount (usually 0.01)
                - Neurons will never die, but they can go 'comatose'
            - Sometimes a small leak is good, apparently often, a large leak is better (0.2 alpha)

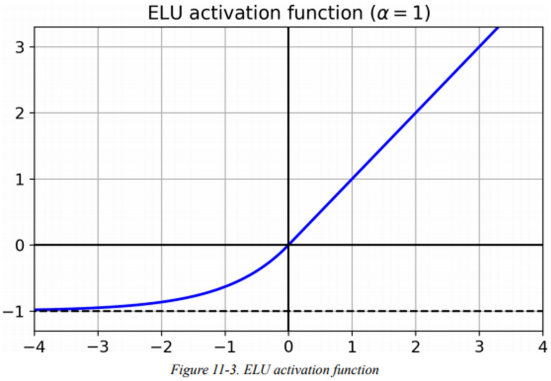
- Randomized Leaky ReLU (RReLU) does pretty well too. It picks alpha at random and then sets it to a fixed average value during the testing period
- Parametric Leaky ReLU (PReLU) where alpha is learned during training
  - Great on large datasets
  - Bad on small ones
- A new activation function in 2015 : https://arxiv.org/pdf/1511.07289.pdf
  - *Exponential Linear Unit (ELU)* outperforming ReLU
  - $\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$
  - 

Figure 11-3. ELU activation function
  - Benefits include:
    - Average output closer to zero when z<0
    - Alpha equaling 1 gives a smooth gradient everywhere, allowing for faster training
    - Nonzero gradient below z=0, no more dying neurons!
    - (con) it is pretty slow though
  - *Scaled ELU (SELU)* https://arxiv.org/pdf/1706.02515.pdf
    - A neural network with only stacked Dense layers with all SELU activations will self-normalize (layer mean 0 and layer standard deviation 1) under certain conditions:
      - Inputs must be standardized
      - LeCun initialization must be used for hidden layer weights
      - Networks must be sequential
      - Supposedly some CNNs with other than dense layers work well too

- Batch Normalization : https://arxiv.org/pdf/1502.03167.pdf
  - Operate on stuff before and after each layer to reduce even further the likelihood of gradient vanishing/exploding issues
    - Zero-center and normalize each input
    - Then rescale and offset them
  - A BN layer will standardize for you
  - How do it do?

- *Equation 11-3. Batch Normalization algorithm*

$$1.\quad \boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$$

$$2.\quad \boldsymbol{\sigma}_B^{\,2} = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

$$3.\quad \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^{\,2} + \varepsilon}}$$

$$4.\quad \mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

  - $\boldsymbol{\mu}_B$ is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).
  - $\boldsymbol{\sigma}_B$ is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
  - m_B is the number of instances in the mini-batch.
  - $\hat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance i.
  - $\boldsymbol{\gamma}$ is the output scale parameter vector for the layer (it contains one scale parameter per input).
  - $\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
  - $\boldsymbol{\beta}$ is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
  - ε is a tiny number that avoids division by zero (typically 10^{-5}). This is called a *smoothing term*.
  - $\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.

        - Better to batch after layer or before (there is debate about this)
        - Tweakable hyperparameters for batch normalization
            - Momentum
                - $\hat{\mathbf{v}} \leftarrow \hat{\mathbf{v}} \times \text{momentum} + \mathbf{v} \times (1 - \text{momentum})$
                - Keep momentum close to 1 (0.9, 0.999, 0.9999…)
                    - The more 9's the better for larger datasets
            - Axis
                - -1 defaults to last axis normalized
                - Which axis to normalize upon
        - Perhaps BN ain't necessary tho???
            - https://arxiv.org/pdf/1901.09321.pdf
    - Gradient Clipping
        - Gradients can never exceed a certain level
        - Good for RNNs
    - Reusing Pretrained Layers
        - Why reinvent the wheel?
            - *Transfer learning*, reusing pieces of good networks
        - Fix the weights of sum layers while only training the others
        - Attempt freezing/unfreezing various reused layers and see how things perform
    - Unsupervised Pretraining
        - Good when there's not enough labeled data
        - Typically used with autoencoders, GANs (?), or less likely restricted Boltzmann machines (RBMs)
        - Greedy layer-wise pretraining
            - Train unsupervised model with one layer (RBM usually)
            - Freeze previous layer and add a new one on top
            - Repeat
        - Pretraining on an Auxiliary Task
            - Train on a dataset that you can easily make more training data of before going to the other data
            - Good for Natural Language Processing (just get more books)
- Faster Optimizers
    - Momentum Optimization
        - Gradient descent with consciousness regarding the previous gradient, thereby building up or losing momentum as slopes are traversed
            1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$
            2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$
    - Nesterov Accelerated Gradient
        - Measure gradient slightly ahead
            1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta} + \beta\boldsymbol{m})$
            2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

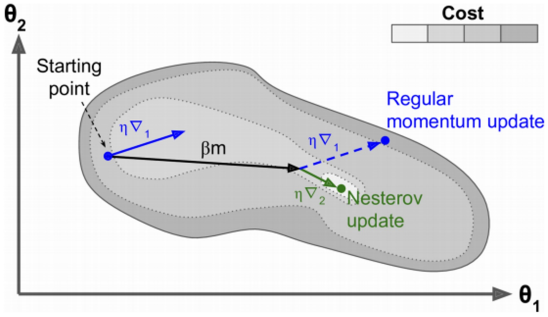
- 

Figure 11-6. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after

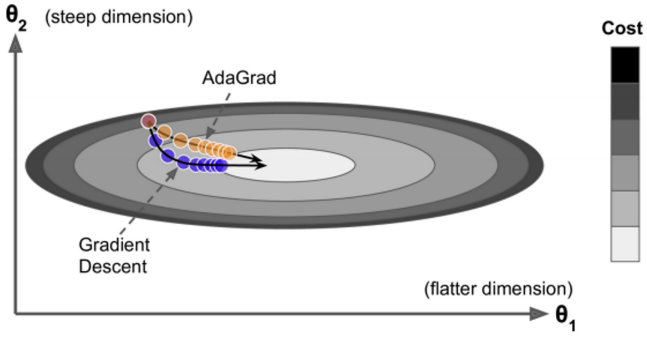
- AdaGrad
  - Scale the gradient vector down along the steepest dimensions
    1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
    2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \varepsilon}$
  - 

Figure 11-7. AdaGrad versus Gradient Descent: the former can correct its direction earlier to point to the optimum
  - Utilizes an adaptive learning rate
- RMSProp
  - Slowing down too fast with AdaGrad? Introducing RMSProp, using only the most recent gradients instead of all of them
    1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
    2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \varepsilon}$
- Adam and Nadam Optimization
  - Adam: *adaptive moment estimation* = momentum optimization and RMSProp
    - Exponentially decaying average of past gradients and past squared gradients
    1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
    2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
    3. $\hat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - \beta_1^{\mathrm{T}}}$
    4. $\hat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - \beta_2^{\mathrm{T}}}$
    5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$
  - Adam variants
    - AdaMax: Adam without steps 3 and 4. Parameter updates scaled by the l_2 norm, except now it's the l_∞ which is the max
    - Nadam: Nesterov Adam
  - Note: sometimes adaptive algorithms don't generalize well
  - NOTE: All optimization algorithms so far have been based on first-order partial derivatives (Jacobians) and many great ones rely on greater order ones (second-order: Hessians)
- Learning Rate Scheduling
  - Adjust the learning rate over time according to some scheduling algorithm
  - Power Scheduling
    - Learning rate is a function of the iteration number, usually to some power

        - $\eta(t) = \frac{\eta_0}{\left(1+\frac{t}{s}\right)}$
    - Exponential Scheduling
        - $\eta(t) = \eta_0 0.1^{t/s}$
    - Piecewise constant scheduling
        - Constant learning rate for a number of epochs, and adjust during the next set of epochs to a different number
    - Performance scheduling
        - Validation error influences the learning rate. Reduce learning rate by factor of λ when error stops dropping
    - 1cycle Scheduling: https://arxiv.org/pdf/1803.09820.pdf
        - Increase learning rate linearly at first up to some constant, and then linearly back down
- Avoiding Overfitting Through Regularization
    - l_1 and l_2 Regularization
        - Constrain weights like we did in Chapter 4
    - Dropout
        - At each step, any neuron (except output neurons) has a chance of being dropped out of training with probability 'p'
        - Monte Carlo Dropout
            - Apparently Drop out is really close to Bayesian inference. Cool
            - Increase performance of trained dropout network without training
            - Monte Carlo sample the model and predict a bunch of samples
    - Max-Norm Regularization
        - We did this before too
- Summary

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

- If you need a sparse model, you can use ℓ_1 regularization (and optionally zero out the tiny weights after training). If you need an even sparser model, you can use the TensorFlow Model Optimization Toolkit. This will break self-normalization, so you should use the default configuration in this case.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use fewer layers, fold the Batch Normalization layers into the previous layers, and possibly use a faster activation function such as leaky ReLU or just ReLU. Having a sparse model will also help. Finally, you may want to reduce the float precision from 32 bits to 16 or even 8 bits (see "Deploying a Model to a Mobile or Embedded Device"). Again, check out TF-MOT.
- If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC Dropout to boost performance and get more reliable probability estimates, along with uncertainty estimates.

Chapter 12 Custom Models and Training with Tensorflow

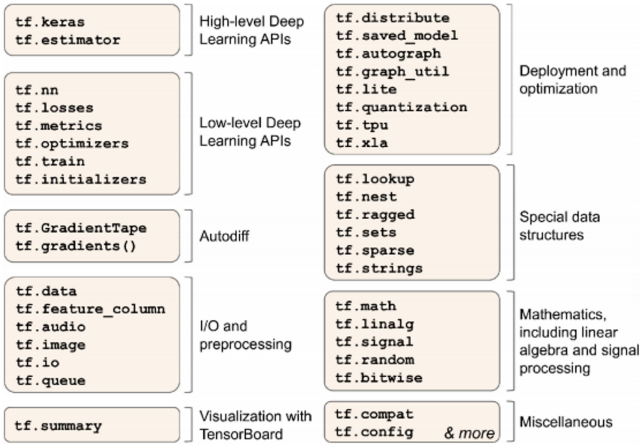
- Low-level Python API
- Tensorflow Grand Tour
  - GPU support
  - Distributed computing available
  - Just-in-time (JIT) compiler allowing optimized computations for speed and memory
    - Extract "*computation graph*" from Python function, optimize it by trimming, and running it efficiently with independent ops in parallel
      - These can be exported to portable medium and used in various other languages (Python on Linux or Java on Android for instance)
  - Provides other nice optimizers (autodiff, RMSProp, Nadam)
  - 

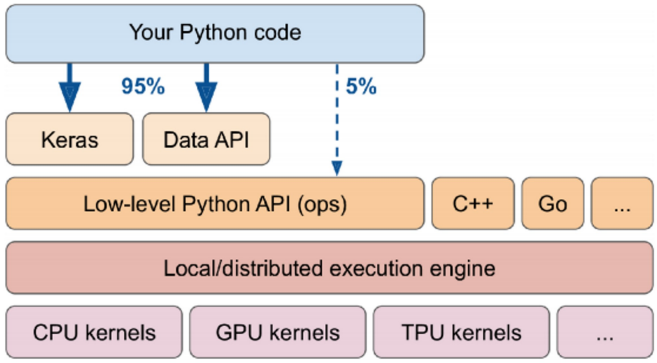
    Figure 12-1. TensorFlow's Python API
  - Under the hood
    - Ops are implemented in C++ with various implementations (kernels)
      - These kernels operate on different hardware types (CPU, GPU, TPU)
    - 

    Figure 12-2. TensorFlow's architecture
    - TF can pretty much run anywhere (Windows, Linux, macOS, Android, iOS) and any how (APIs exist for Python, C++, Java, Go, Swift, and even JavaScript)
    - Is the core of other Tensor products

        - TensorBoard : visualization
        - TensorFlow Extended (TFX) : to productionize TF with data validation, preprocessing, model analysis, and serving
        - TensorFlow Hub : easy download and resuse of pretrained NN
            - TensorFlow's Model Garden : pretrained NNs
    - Many resources using TF : https://paperswithcode.com/
- Using TensorFlow like NumPy
    - Tensors flowing from op to op
        - Tensor: multidimensional array
        - You can add them, transpose them, square them, you name it!
        - Operations are equivalent to function calls
            - Keras also has functions for these as well
    - Tensorflow works well with NumPy arrays
    - Incompatible types will bring about exceptions since type conversions are really inefficient
        - Float + interger = no
        - 32-bit + 64-bit = no
        - Use 32-bits
    - Tensors aren't mutable, making them bad for storing weights and whatnot, so use tf.Variable
        - Use 'assign' functions to update this
        - Keras has an 'add_weight()' function that takes care of most variables
    - Other Data Structures
        - Sparse Tensors: mostly zeros
        - Tensor Arrays: lists of tensors with same shape and data type
        - String Tensors: byte strings (not Unicode
        - Sets: self-explanatory
        - Queues: keeping tensors saved during multiple operations
            - FIFO, PriorityQueue, RandomShuffleQueue, PaddingFIFOQueue
- Customizing Models and Training Algorithms
    - Custom Loss Functions
        - Function definition that takes two tensors of the same size and reports on them some error
    - Saving and Loading Models that Contain Custom Components
        - Gotta finagle with some class inheritance to save proper variables and whatnot
        - Use dictionary mapping to save hyperparameters and other customizables
        - Saving a model that implements get_config will save said parameters along in the JSON format
    - Custom Activation Functions, Initializers, Regularizers, and Constraints
        - Pretty simple too, same issue with inputs and outputs too,
        - Wrap them in a class and implement get_config
        - "Note that you must implement the call() method for losses, layers (including activation functions), and models, or the __call__() method for regularizers, initializers, and constraints. For metrics, things are a bit different, as we will see now."
    - Custom Metrics
        - For evaluation and not training
        - Default is keeping track of the average over an epoch, but that can be changed
        - For the case of keeping track of Precisions, try using Keras's Precision class
        - Streaming metrics are cool!, subclass keras Metric
    - Custom Layers
        - No weight layers (flatten, ReLU) can be wrapped in Keras Lambda
        - Other layers
            - Subclass the keras Layer class
            - Layers that have different properties during training and testing

- Custom Models
  - Subclass and implement call, that's pretty much it
  - Notebook contains an implementation of this model:

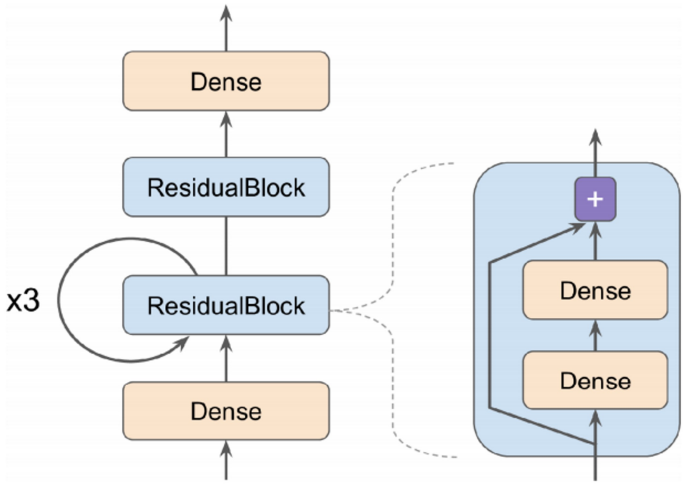


Figure 12-3. Custom model example: an arbitrary model with a custom ResidualBlock layer containing a skip connection

    - Nonsensical but has loops and skips
- Losses and Metrics Based on Model Internals
  - Using information of the model to influence metrics and whatnot
- Computing Gradients Using Autodiff (how TF does it)
  - Adjust input slightly by some small epsilon and interpolate linearly

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

  - Calling this multiple times is naughty
    - Specify your GradientTape as being persistent=True and that should fix it, but then make sure to delete it!

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # => tensor 10.0, works fine now!
del tape
```

  - Note it also only works with variables
    - Unless you watch tensors

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [tensor 36., tensor 10.]
```

  - Use *jacobian* for reverse-mode gradients for Hessian
  - Stop gradients from backpropagating!
  - Sometimes numerical errors pop up like with undefined gradients, for this you can use a custom gradient and compute it analytically

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

- Custom Training Loops
  - When the 'fit' function just isn't fit for the job

- TensorFlow Functions and Graphs
  - Can transform python functions into TF functions
  - TF functions generate a single graph for each non-TF variable or creates one per tensor of a particular shape
  - Autograph and Tracing
    - Basically takes a look at all control loops and conditionals and optimizes them

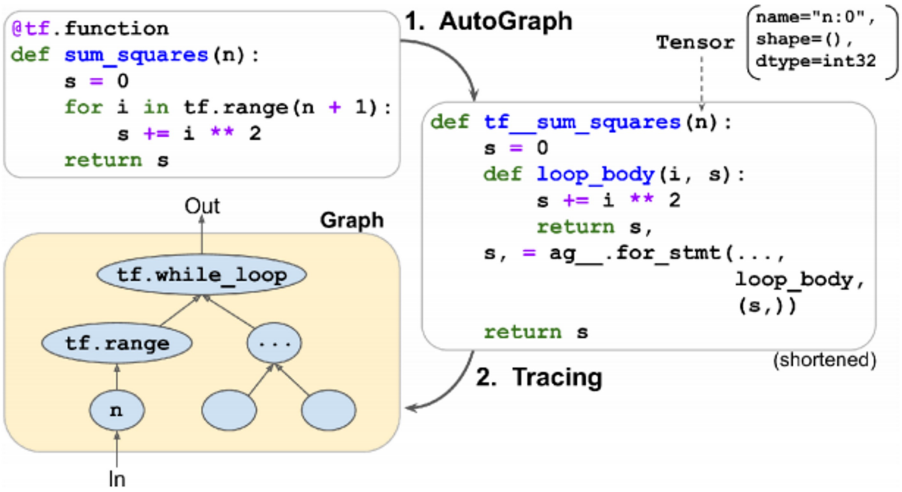


Figure 12-4. How TensorFlow generates graphs using AutoGraph and tracing

    - Runs in Graph Mode as opposed to eager mode
  - TF Function Rules
    - Use TF functions and not external ones like from NumPy (if you do, then it only gets called during the tracing step)
      - Wrapping said code in tf.py_function() will block TF from optimizing it
    - If a function creates a TF variable it must do so on the first call, or else exceptions occur
    - Source code should be available
    - For loops only get optimized if they are done over a tf tensor or dataset
    - Use vectorized implementations when possible

Chapter 13 Loading and Preprocessing Data with Tensorflow

- What if your datasets don't fit in RAM? Use the Data API
  - Give location of data, what to do with it and then TF takes care of manipulation
  - Types of Data include: CSVs, binary files with fixed-sized records, TFRecord format files based on Protocol Buffers, SQL databases, and others with Google's BigQuery service
- Problems
  - Big data numbers
  - Proper normalization
    - Encoding for categorical data
- Solutions
  - Code own preprocessing layers
  - Keras's preprocessing layers
- The Data API

- Basic manipulation
- Shuffling
  - On large data, it might be better to shuffle the data ahead of time (like Unix's shuf)
  - Perhaps split data into multiple files and read in a random order
  - Interleaving files
    - ```
      filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
      ```
      - Default shuffles, set shuffle to False if you don't want this
    - ```
      n_readers = 5
      dataset = filepath_dataset.interleave(
          lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
          cycle_length=n_readers)
      ```
      - Add num_parallel_calls for multithreading
        - Make equal to tf.data.experimental.AUTOTUNE for optimzied threading
    - Ensure that files are of equal length or else those ends will not get interleaved
- Preprocessing the Data
- Prefetching
  - Caching some batches ahead of time
    - There's a 'cache' function for that too
  - Other functions of data storage and whatnot
    - Concatenate, zip, window, reduce, shard, flat_map, padded_batch, from_generator, from_tensors, CsvDataset.make_csv_dataset
- Using the Dataset with tf.keras
  - ```
    train_set = csv_reader_dataset(train_filepaths)
    valid_set = csv_reader_dataset(valid_filepaths)
    test_set = csv_reader_dataset(test_filepaths)
    model = keras.models.Sequential([...])
    model.compile([...])
    model.fit(train_set, epochs=10, validation_data=valid_set)
    model.evaluate(test_set)
    new_set = test_set.take(3).map(lambda X, y: X) # pretend we have 3 new instances
    model.predict(new_set) # a dataset containing new instances
    ```
  - TF Function for an entire training loop
    - ```
      @tf.function
      def train(model, optimizer, loss_fn, n_epochs, [...]):
          train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
          for X_batch, y_batch in train_set:
              with tf.GradientTape() as tape:
                  y_pred = model(X_batch)
                  main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
                  loss = tf.add_n([main_loss] + model.losses)
              grads = tape.gradient(loss, model.trainable_variables)
              optimizer.apply_gradients(zip(grads, model.trainable_variables))
      ```

- The TFRecord Format
  - Creating an easy record
    - ```
      with tf.io.TFRecordWriter("my_data.tfrecord") as f:
          f.write(b"This is the first record")
          f.write(b"And this is the second record")
      ```
  - Reading one or multiple files
    - ```
      filepaths = ["my_data.tfrecord"]
      dataset = tf.data.TFRecordDataset(filepaths)
      for item in dataset:
          print(item)
      ```
  - Compressed TFRecord Files

- Create them
  - ```
    options = tf.io.TFRecordOptions(compression_type="GZIP")
    with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
      [...]
    ```
- Reading them
  - ```
    dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                      compression_type="GZIP")
    ```
- A Brief Introduction to Protocol Buffers
  - Kinda looks like this:
    - ```
      syntax = "proto3";
      message Person {
        string name = 1;
        int32 id = 2;
        repeated string email = 3;
      }
      ```
    - .proto files compiled with *protoc*
- TensorFlow Protobufs
  - Kinda looks like:
    - ```
      syntax = "proto3";
      message BytesList { repeated bytes value = 1; }
      message FloatList { repeated float value = 1 [packed = true]; }
      message Int64List { repeated int64 value = 1 [packed = true]; }
      message Feature {
          oneof kind {
              BytesList bytes_list = 1;
              FloatList float_list = 2;
              Int64List int64_list = 3;
          }
      };
      message Features { map<string, Feature> feature = 1; };
      message Example { Features features = 1; };
      ```
- Loading and Parsing Examples

○ Preprocessing the Input Features

- ```
  means = np.mean(X_train, axis=0, keepdims=True)
  stds = np.std(X_train, axis=0, keepdims=True)
  eps = keras.backend.epsilon()
  model = keras.models.Sequential([
      keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
      [...] # other layers
  ])
  ```
- OR

  ```
  class Standardization(keras.layers.Layer):
      def adapt(self, data_sample):
          self.means_ = np.mean(data_sample, axis=0, keepdims=True)
          self.stds_ = np.std(data_sample, axis=0, keepdims=True)
      def call(self, inputs):
          return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
  ```
- Standardize
  - ```
    std_layer = Standardization()
    std_layer.adapt(data_sample)
    ```
- Use preprocessing layer as normal layer
  - ```
    model = keras.Sequential()
    model.add(std_layer)
    [...] # create the rest of the model
    model.compile([...])
    model.fit([...])
    ```
- Encoding Categorical Features Using One-Hot Vectors
- Encoding Categorical Features Using Embeddings
  - Can be trained
  - Pretty cool
- Keras Preprocessing Layers
    ```

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

- TF Transform from TFX
- TF Datasets