Training ML models

Cambridge ICCS summer school

cambridge-iccs.github.io/summerschool

Will Handley

2022-09-22

Slides & installation instructions at

`https://github.com/handley-lab/2022-cambridge-iccs`

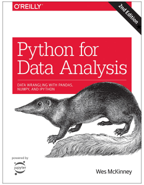
Overview

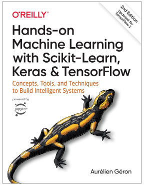
- Title is rather broad, and we have < 2h, the morning after the conference dinner.
- Aim to build a framework of understanding in the context of a few examples
- Should be able to answer:
  - What is an ML model?
  - How do I avoid pitfalls in training them?
  - Which resources should I reach for in the future?

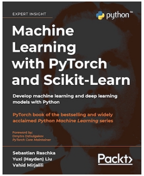
The Machine Learning Python stack

0. `numpy`
   - Layer zero – vector maths & array-based programming
   - Advanced users: broadcasting, `x @ y`, `z[:,None]`, ufuncs
1. `scipy` & `pandas`
   - Extends `numpy` to numerical algorithms and excel-like array functionality
   - `pandas` often a data scientist’s weakest point
2. `scikit-learn`
   - Entry-level machine learning
   - Extends to allow estimators, transformers & predictors
   - With a few key concepts this is a consistent and versatile ML framework
3. `Keras`/`TensorFlow` `PyTorch`,
   - Deep learning tools
   - Familiarity with the previous layers greatly enhances effectiveness

+ `matplotlib` for plotting (others exist, extending and reducing flexibility)



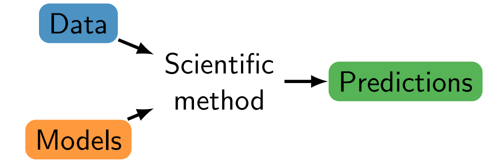

O'REILLY
2nd Edition Updated for TensorFlow 2
Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow
Concepts, Tools, and Techniques to Build Intelligent Systems
powered by jupyter
Aurélien Géron


EXPERT INSIGHT
python
Machine Learning with PyTorch and Scikit-Learn
Develop machine learning and deep learning models with Python
PyTorch book of the bestselling and widely acclaimed Python Machine Learning series
Foreword by: Dmytro Dzhulgakov PyTorch Core Maintainer
Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili
Packt

What is machine learning?

What is machine learning?

- A computer program which can program itself to perform a task
  - Problems with lots of tuning/rules
  - Problems with no traditional solution
  - Fluctuating environments
  - Gaining insight about complex data
- Traditional programs
  - Quicksort
  - Pong
- Machine learning
  - Spam filter
  - Netflix suggestions
  - Speech recognition
  - Dall-E

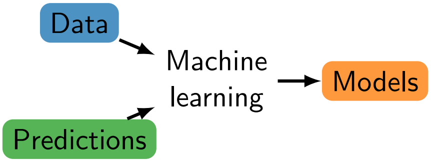


Data

Models

Scientific method

Predictions



Data

Predictions

Machine learning

Models

The machine learning pipeline

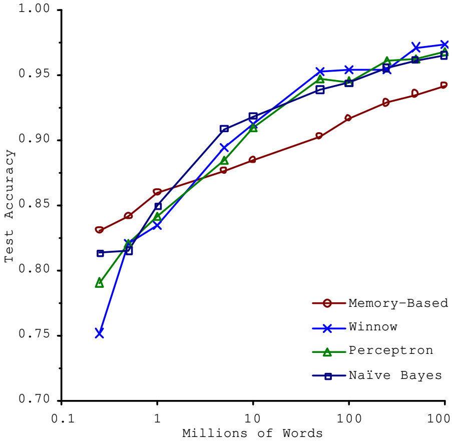
1. Problem framing
2. Data acquisition
3. Visualisation
4. Data preparation/munging
5. Selecting & training a model
6. Tuning a model
7. Launch, monitor & maintain

The machine learning pipeline

1. Problem framing
   - Big picture
   - Selecting performance measures/objectives
   - Checking assumptions/bias
2. Data acquisition
3. Visualisation
4. Data preparation/munging
5. Selecting & training a model
6. Tuning a model
7. Launch, monitor & maintain

The machine learning pipeline

1. Problem framing
2. Data acquisition
   - Gather your data
   - Selecting performance measures/objectives
   - Checking assumptions/bias
3. Visualisation
4. Data preparation/munging
5. Selecting & training a model
6. Tuning a model
7. Launch, monitor & maintain



[doi:10.3115/1073012.1073017]

The machine learning pipeline

1. Problem framing
2. Data acquisition
3. Visualisation
    - using pandas+matplotlib skills to explore ideosyncrasies of the data
4. Data preparation/munging
5. Selecting & training a model
6. Tuning a model
7. Launch, monitor & maintain

The machine learning pipeline

1. Problem framing
2. Data acquisition
3. Visualisation
4. Data preparation/munging
   - using scikit-learn to clean & transform the data
5. Selecting & training a model
6. Tuning a model
7. Launch, monitor & maintain

The machine learning pipeline

1. Problem framing
2. Data acquisition
3. Visualisation
4. Data preparation/munging
5. Selecting & training a model
    - Topic of this session
6. Tuning a model
7. Launch, monitor & maintain

The machine learning pipeline

1. Problem framing
2. Data acquisition
3. Visualisation
4. Data preparation/munging
5. Selecting & training a model
6. Tuning a model
   - Topic of this session
7. Launch, monitor & maintain

The machine learning pipeline

1. Problem framing
2. Data acquisition
3. Visualisation
4. Data preparation/munging
5. Selecting & training a model
6. Tuning a model
7. Launch, monitor & maintain
    - For researchers this could be github distributing
    - For industry this would mean real-world shipping

Categories of machine learning

Supervised

Regression
Classification

Unsupervised

Clustering, Visualisation, Dimensionality reduction

Semisupervised

Google Photos

Reinforcement

AlphaGo, GANNS

Categories of machine learning

Supervised

Regression
Classification

Unsupervised

Clustering, Visualisation, Dimensionality reduction

Semisupervised

Google Photos

Reinforcement

AlphaGo, GANNS

Batch

Offline learning using all available data.

Online

Training/updating on-the-fly on mini-batches, for memory-bound/out-of-core

Categories of machine learning

Supervised

Regression
Classification

Unsupervised

Clustering, Visualisation, Dimensionality reduction

Semisupervised

Google Photos

Reinforcement

AlphaGo, GANNS

Batch

Offline learning using all available data.

Online

Training/updating on-the-fly on mini-batches, for memory-bound/out-of-core

Instance based learning

"Learn-by-heart" – given a similarity measure, compare/regress/classify new examples onto existing data

Model-based learning

Build a parameterised model of data, train it, then make predictions

Challenges of Machine Learning

- Bad data
  - Not enough
  - Not representative
  - Poor quality (outliers, noise)
  - Irrelevant/Poor features
- Bad algorithm/Bad training (Focus of this workshop)
  - Overfitting
  - Underfitting

Example 1: Introduction to regression

Slides & installation instructions at
`https://github.com/handley-lab/2022-cambridge-iccs`

Training an ML model

- Three ingredients to training
  1. Input data/features $\boldsymbol{x}^{(i)}$, output data $y^{(i)}$, where $i = 1, \ldots n_{\text{obs}}$
  2. Parameterised model $\boxed{y = h_{\boldsymbol{\theta}}(\boldsymbol{x})}$, where h is the model and $\boldsymbol{\theta}$ are its parameters
  3. Loss function(s) $L(y_{\text{pred}}, y)$
- Train the parameters on a training subset by solving the mathematical problem

$$\hat{\boldsymbol{\theta}} = \min_{\boldsymbol{\theta}} \sum_{i \in \text{train}} L_{\text{train}} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)} \right)$$

- Choose the best model by minimising a (possibly different) loss on a validation subset

$$\hat{h} = \min_{h} \sum_{i \in \text{validation}} L_{\text{validation}} \left(h_{\hat{\boldsymbol{\theta}}}(\boldsymbol{x}^{(i)}), y^{(i)} \right)$$

- Finally test the best model on set-aside testing data.

1. Data

- The initial data $\{(\boldsymbol{x}^{(i)}, y^{(i)}), i = 1, \ldots, n_{\text{obs}}\}$ must usually be transformed
- Relevant features should be selected
- Relevant combinations of features should be considered (e.g. computing rates/sums)
- This is known as feature engineering
- Features should then be normalised, Either:

  min-max scaled data lie in $[0, 1]$ (`sklearn.preprocessing.MinMaxScaler`)
  standardised data have mean 0 and std 1 (`sklearn.preprocessing.StandardScaler`)
- The `sklearn` way to do this is to chain a set of these transformations together in a `sklearn.pipeline.Pipeline`
- This is essential since almost all machine learning algorithms are not covariant, and will fail on unnormalised data.

2. Models

- "How to choose models" would fill a whole other session.
- You may recognise some of the standard choices.
- Models have trainable parameters $\boldsymbol{\theta}$, and hyperparameters.

Supervised

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines
- Decision Trees and Random Forests
- Neural Networks

Semisupervised

- deep belief networks
- RBMs

Reinforcement

- AlphaGo
- GANNs

Unsupervised

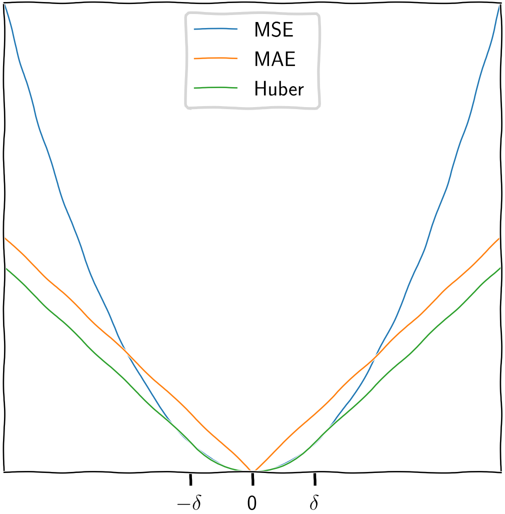
- Clustering
  - K-Means
  - DBSCAN
  - Hierarchical Cluster Analysis
- Anomaly detection
  - One-class SVM
  - Isolation forest
- Visualisation and dimensionality reduction
  - (Kernel) PCA
  - Locally-Linear Embedding
  - t-SNE
- Association rule learning
  - Apriori & Eclat

3. Loss functions

- All that is needed is something which measures how "close" a model's prediction is to the true answer
- The loss function you train on does not need to be the same as the testing/validation metric.
- Mean square error (MSE) $L = (\Delta y)^2$
  - smooth (differentiable)
- Mean absolute error (MAE) $L = |\Delta y|$
  - robust to outliers
- Huber loss

$$L = \begin{cases} \frac{1}{2}(\Delta y)^2 & |\Delta y| < \delta \\ \delta((\Delta y) - \frac{1}{2}\delta) & |\Delta y| >= \delta \end{cases}$$

  - combines benefits of both



Principles of Training, Validation & Testing splits

- The sure-fire way to know how a model will generalise is to hold back data for testing.
- We therefore split data into three categories

Training

∼80% of the data.
Used for learning parameters.

+Validation

Used for learning hyperparameters.

Testing

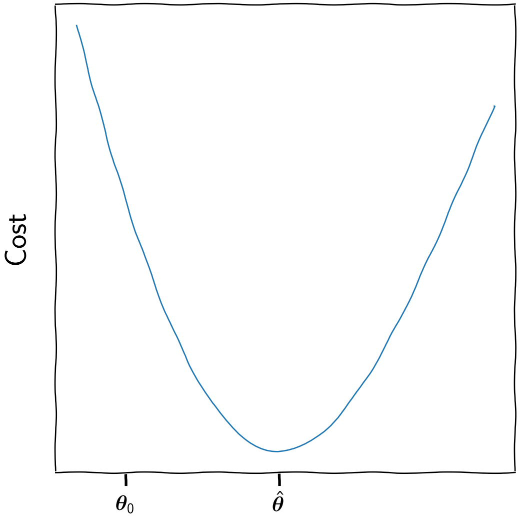
Holdout ∼20% of the data.
Should be ideally "one-shot".

- "Validation set" is also known as the "development/dev set".
- If we do repeated holdout validation on many small validation sets, this is cross-validation.
- Gotchas:
  - standardisation/data preparation should only use the training+validation set
    - Failing to do this reduces generalisability
  - ideally the splitting procedure should be random, but seedable
    - Failing to do this reduces repeatability & reliability
  - `sklearn.model_selection.StratifiedKFold` accomplishes these and more

Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

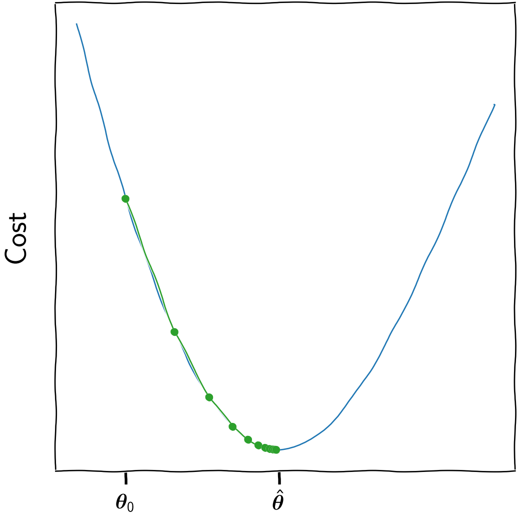
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$



Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

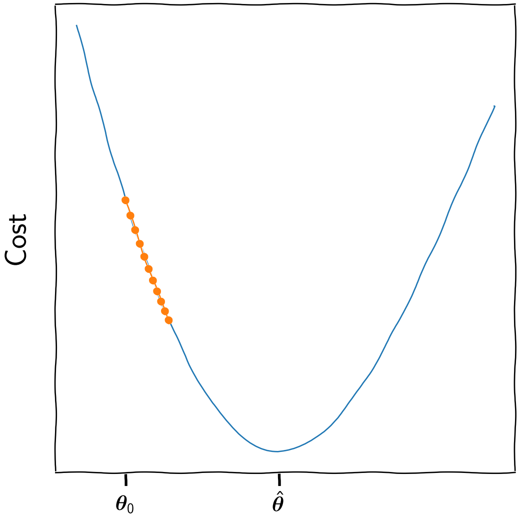


Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

- Problems:
  - Learning rate too slow

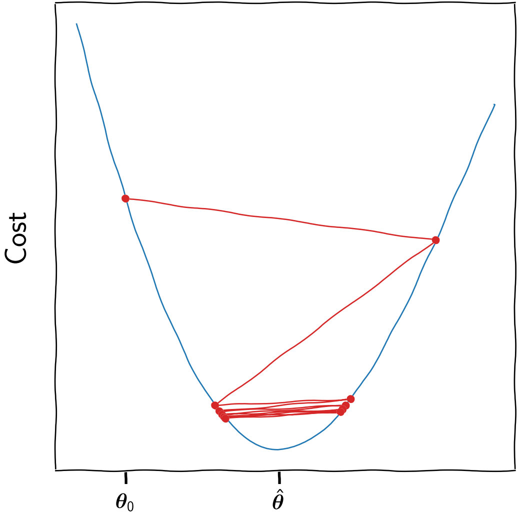


Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

- Problems:
  - Learning rate too slow
  - Learning rate too fast

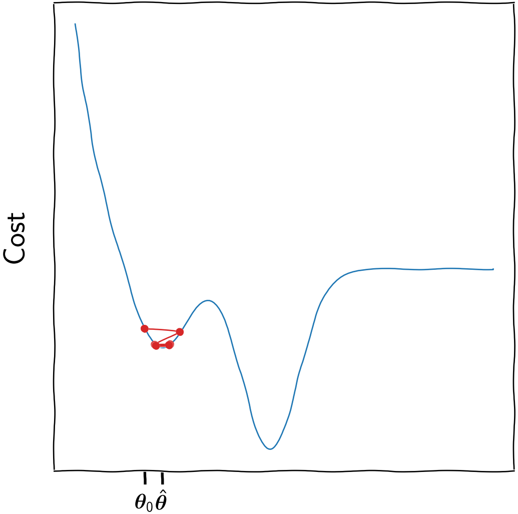


Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

- Problems:
  - Learning rate too slow
  - Learning rate too fast
  - Local minima

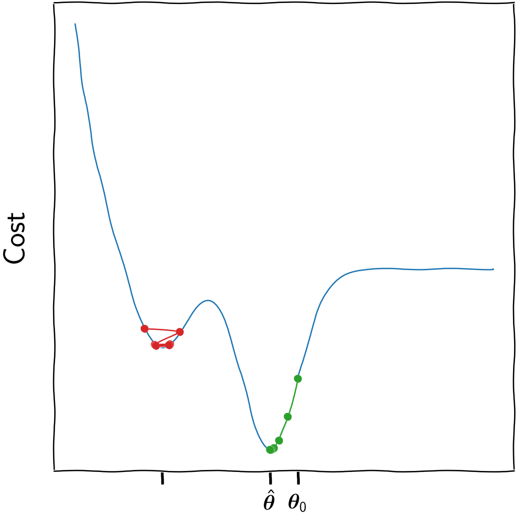


Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

- Problems:
  - Learning rate too slow
  - Learning rate too fast
  - Local minima

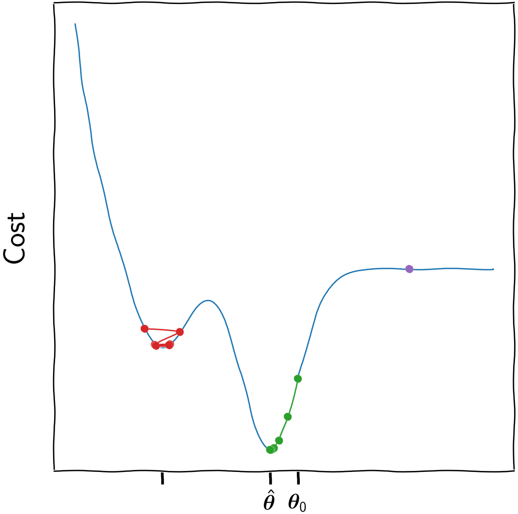


Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

- Problems:
  - Learning rate too slow
  - Learning rate too fast
  - Local minima
  - stalling

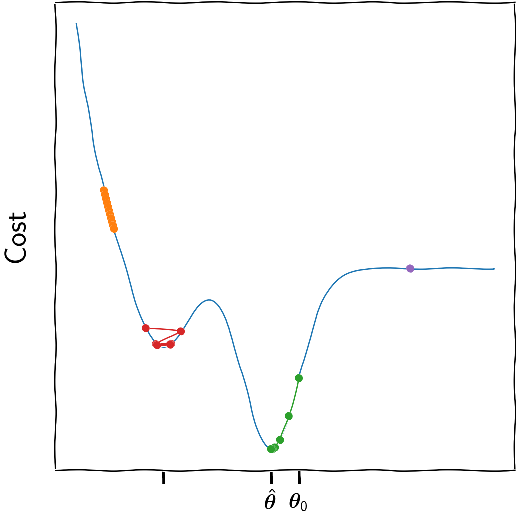


Gradient descent

- To minimise a function $f(\boldsymbol{\theta})$, start somewhere $\boldsymbol{\theta}_0$ and go downhill (down the gradient) by some step η

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

- Problems:
  - Learning rate too slow
  - Learning rate too fast
  - Local minima
  - stalling
- Choosing the learning rate, or more generally tuning the learning schedule can be the hardest part of training.
- All much harder in higher dimensions



Example 2: Machine learning with scikit-learn

Go back to the notebook `training_ml_models.ipynb`

For more detail on the scikit-learn API (beyond the docs) I recommend [arxiv:1309.0238]

How we get the gradients: autodiff

- There are three ways to get a computer to compute a gradient $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$
  1. Analytically
     - Painstakingly coding the function explicitly
     - Accurate, but practically impossible for all but the simplest functions
  2. Numerically
     - Computing finite differences $[\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})]_i \approx [f(\boldsymbol{\theta} + \delta\, \hat{\boldsymbol{e}}_i) - f(\boldsymbol{\theta})]\,/\delta$
     - Easy, but prone to numerical instability
     - Expensive – costs $\sim \mathcal{O}(n)$ function evaluation for each coordinate direction $\hat{\boldsymbol{e}}_i$
  3. Automatically
     - Every computer programme is composed of logic, multiplication and addition.
     - A smart enough computer, equipped with a chain rule, can therefore differentiate any code.
     - Remarkably, this can be done at the same cost as computing f
- Autodiff has been around for a long time, but unless the programmming language is designed with it in mind it is difficult in practice.
- Modern ML codes are.
- In traditional ML literature this is wrapped up in a mythology of "backpropagation" – equivalent to "reverse mode autodiff", well suited to high-chaining with few outputs.

Variations on gradient descent

Batch gradient descent (full)

Compute the gradient in full at each step:

$$\nabla_{\boldsymbol{\theta}} \sum_i \left[L(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}) \right]$$

Deterministic.
Slow for large data sets.

Stochastic gradient descent (SGD)

Compute the gradient on one random data point

$$\nabla_{\boldsymbol{\theta}} \left[L(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(j)}), y^{(j)}) \right]$$

Blazingly fast.
Randomness can escape minima.

Mini-batch gradient descent

Compute the gradient on a (small) subset of points

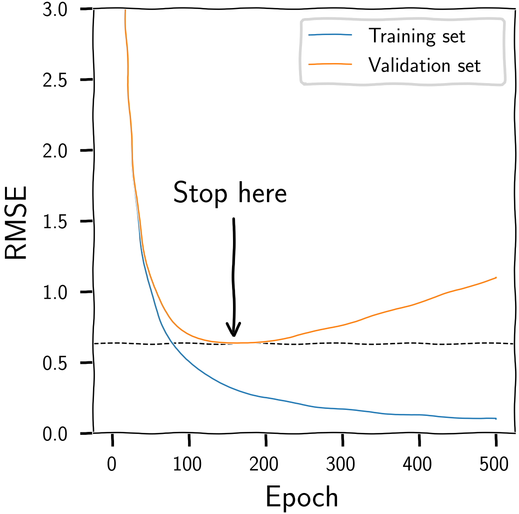
$$\nabla_{\boldsymbol{\theta}} \sum_{i \in \text{batch}_j} \left[L(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}) \right]$$

GPU accelerable.
Batch size < 32.

- In addition SGD & mini-batch have benefits for out-of-core learning, when the data are too large to fit into memory.

Regularisation

- Can reduce overfitting by constraining the degrees of freedom of the model
  - reduce the number of free parameters
  - constrain the parameters to a reduced range
- Regularisation (only applied at training)
  - Ridge regression: $L(\boldsymbol{\theta}) + \frac{1}{2}\alpha\sum_i \theta_i^2$
  - Lasso regression: $L(\boldsymbol{\theta}) + \alpha\sum_i |\theta_i|$
    - Least absolute shrinkage & selection operator
    - Promotes sparsity
  - Elastic Net: $L(\boldsymbol{\theta}) + r\alpha\sum_i |\theta_i| + \frac{1-r}{2}\alpha\sum_i \theta_i^2$
- Early stopping
  - Halt training when validation error increases.
  - As close to a free lunch as one gets.
  - Very popular.



Neural networks

A multi-layer perceptron (MLP) is mathematically

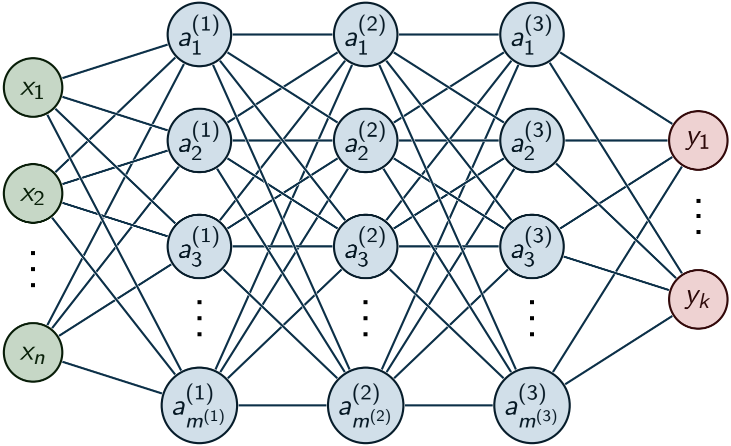
$$a_i^{(\ell+1)} = \phi_i^{(\ell)}\left(\sum_{j=1}^{m^{(\ell)}} w_{ij}^{(\ell)} a_j^{(\ell)} + b_i^{(\ell)}\right)$$

or written in vectors:

$$\boldsymbol{a}^{(\ell+1)} = \boldsymbol{\Phi}^{(\ell)}\left(\boldsymbol{W}^{(\ell)}\boldsymbol{a}^{(\ell)} + \boldsymbol{b}^{(\ell)}\right)$$

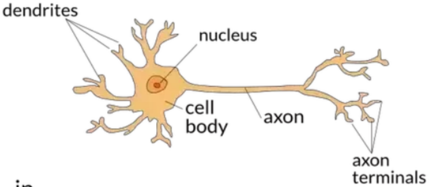
$$\boldsymbol{a}^{(0)} = \boldsymbol{x}, \qquad \boldsymbol{y} = \boldsymbol{a}^{(m+1)}$$

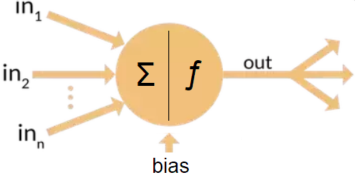
or graphically:

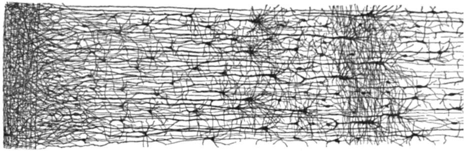


Neural networks & “deep” learning

- NNs were originally inspired by biology.
- Old technology (1960s), came of age in 1990s.
  - rise in computing power
  - innovations in training
  - funding/interest from tech companies
- Universal approximation theorem:
  Any function $\mathbb{R}^n \to \mathbb{R}^k$ can be approximated by a sufficiently wide single hidden layer NN
- So why do we need “deep” (multi-layer) networks?
- Earlier layers perform feature learning to pipe into final universal approximating layer
- Enables the rudiments of transfer learning







Deep learning tools

TensorFlow (Google/Alphabet)

- 2015
- Symbolic math library
- Keras makes easier
- More popular in industry

PyTorch (Facebook/Meta)

- 2017
- Easier to get started
- Faster than Keras
- More popular in research

- Keras is a Python API to TensorFlow, CNTK & Theano
- CNTK is MicroSoft's (now defunct) "cognitive toolkit"
- Theano used to be a giant in the field, and is a 2007 "grandfather" to the rest. Now only used by research/legacy code

In summary: the big boys & girls in industry use TensorFlow, but since PyTorch is preferred and developed by research now, it may become dominant in a few years time. Keras is a wrapper to TensorFlow which makes it more native, but slower and less flexible

Neural Network Anatomy & Training

The dials you can twiddle

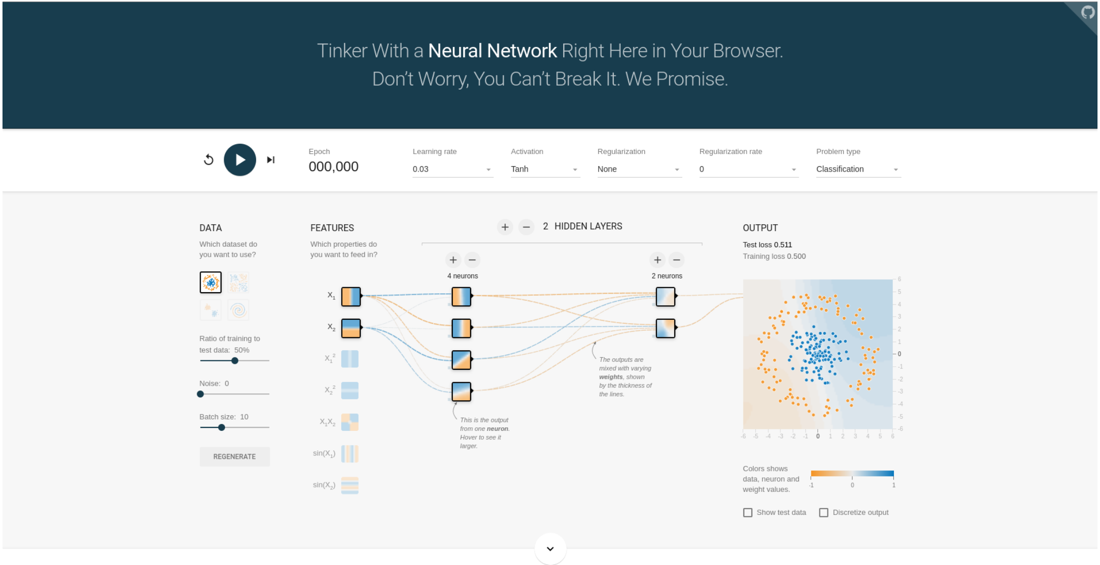
Anatomy

- Number of hidden layers
- Width of hidden layers
- Activation functions

Training

- Loss function
- Optimiser
- Initialisation
- Normalisation
- Regularisation
- Learning rate schedule

playground.tensorflow.org



Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons

2 neurons

*This is the output from one **neuron**. Hover to see it larger.*

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

OUTPUT

Test loss 0.511

Training loss 0.500

Colors shows data, neuron and weight values.

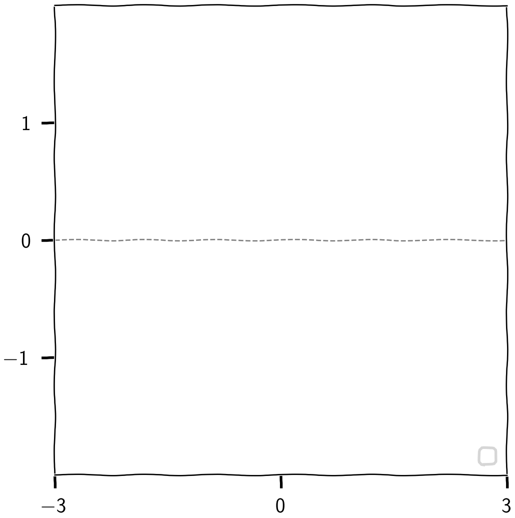
□ Show test data □ Discretize output

Hyperparameter tuning

- The amount of things you can change in a neural network is both an advantage and the main drawback
- Coming up with new combinations forms a thriving field in itself
- There are some automated tools for hyperparameter tuning beyond the cross-validation grids we've found before
  - Hyperopt (Python)
  - Hyperkeras, kopt or talos (Keras)
  - Scikit-Optimise (e.g. `BayesSearchCV`)
  - Spearmint (more Bayesian optimisation)
  - Sklearn-Deap (evolutionary algorithms)
  - . . .
- There are also cloud computing services (Google Cloud, Arimo, SigOpt, Oscar,. . .)
- Finding a more principled way to choose these parameters is a big unsolved problem.
- Nevertheless, there are still some principles which are helpful to know

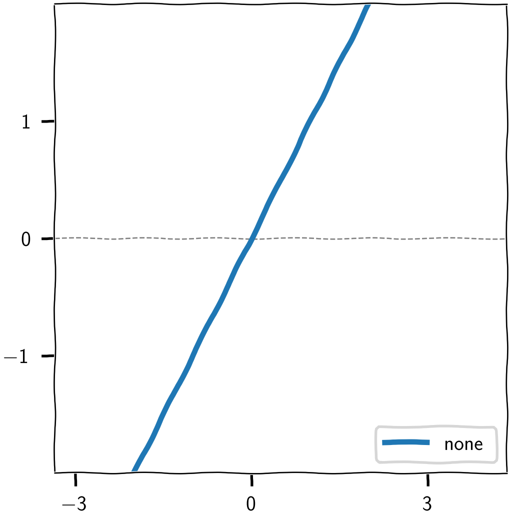
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



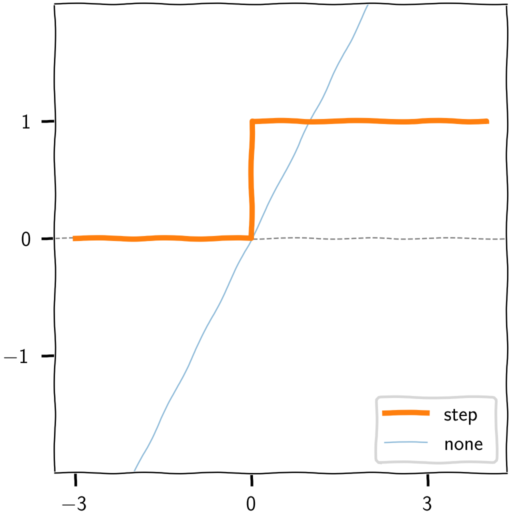
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



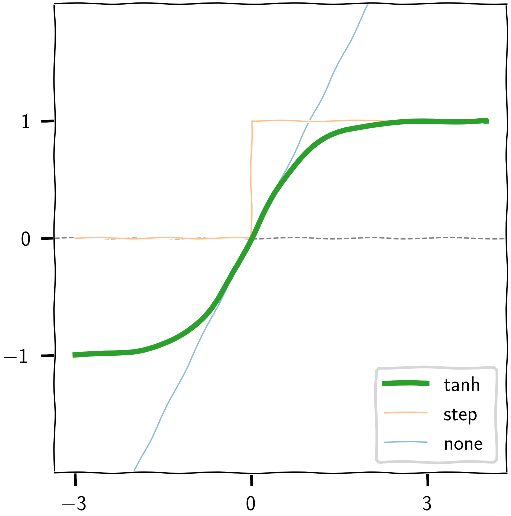
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



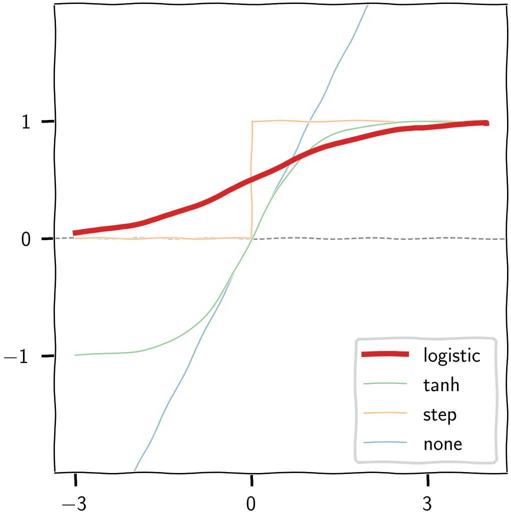
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



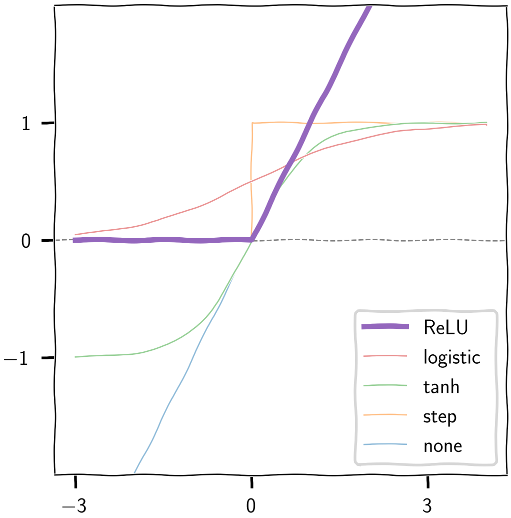
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



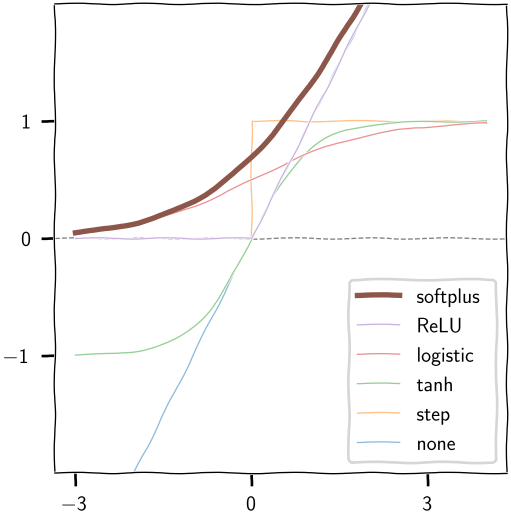
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
    - Smoothness
    - Saturation (at either end)
- Different roles depending on layer
    - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
    - For inner layers one may want symmetry/infinite range



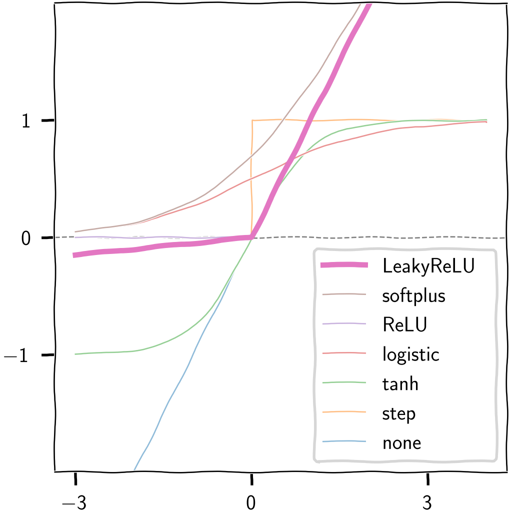
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



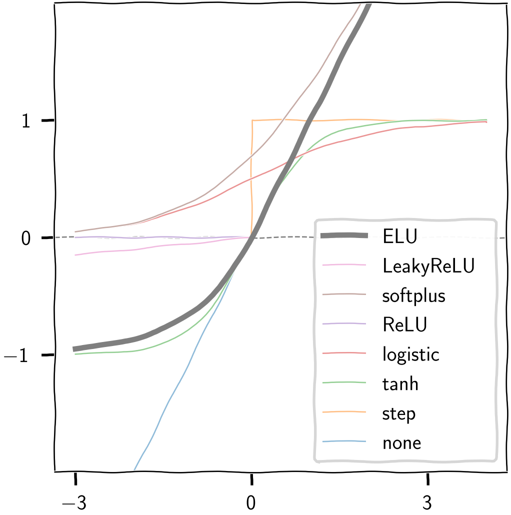
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



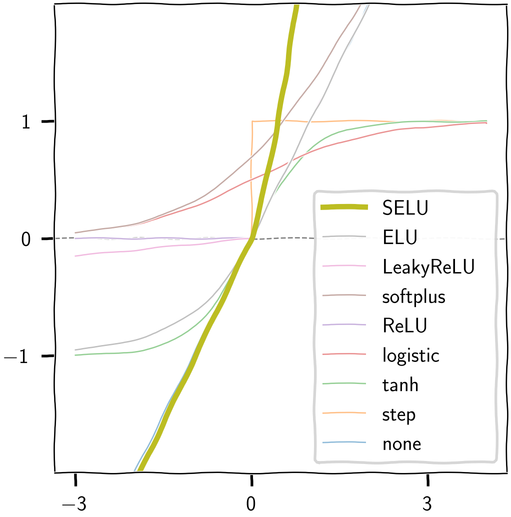
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



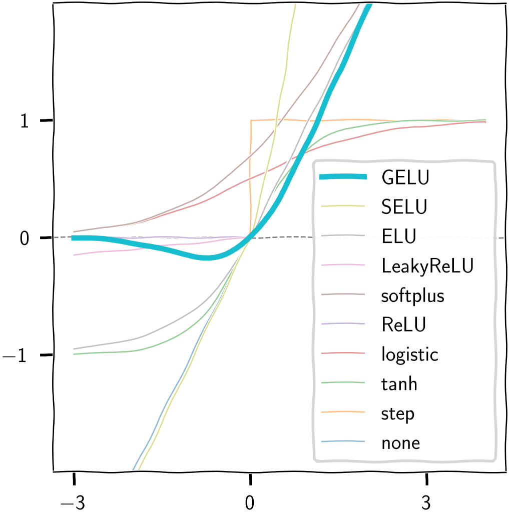
Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



Activation functions

- There is now a veritable zoo of activation functions
- Important properties to consider:
  - Smoothness
  - Saturation (at either end)
- Different roles depending on layer
  - At output it is useful to e.g. impose positivity with softmax or [0,1] boundedness with logistic
  - For inner layers one may want symmetry/infinite range



Typical architectures

All problem dependent, but some reasonable guidelines are

Regression

# input neurons	1 per input feature
# hidden layers	1 to 5
# neurons per hidden layer	Typically 10 to 100
# output neurons	1 per output dimension
Hidden activation	ReLU (or SELU)
Output activation	None
(positive outputs)	ReLU/Softplus
(bounded outputs)	Logistic/Tanh
Loss function	MSE
(if outliers)	MAE/Huber

Classification

Same as Regression, except

- Loss function: cross-entropy
- # output neurons: same as number of labels/classes
- Output layer activation: Logistic for binary classifications and softmax for multiclass.

Number of hidden layers

- Often a single layer will do (UAP)
- Deep networks allow you to do feature learning in the earlier layers
- This also enables transfer learning
- Start with one or two hidden layers, and gradually ramp up until you start overfitting.
- It may be helpful to use pre-trained networks

Width of hidden layers

- Historically we “ramped down”, e.g. starting with 300, then 200, then 100
- In practice this makes little difference and adds tuning parameters
- Rectangular networks therefore more common
- Start with a small number and ramp up until the model starts overfitting

Building neural networks

- Sequential API
  - Straightforward models
- Functional API
  - Complicated models
- Subclassing API
  - Dynamic models

Example 3: Training Neural Networks

Go back to the notebook `training_ml_models.ipynb`

Difficulties in training deep networks

- Exploding/vanishing gradients
- Not enough training
- Slow training
- Overfitting due to to many parameters

Vanishing/Exploding gradients: weight initialisation

- Gradients vanish at plateaus, and explode if able to grow without bound (more common in recurrent neural networks)
- A step-change improvement in performance can be found by weight initialisation ($\boldsymbol{\theta}_0$)
- Standard normally distributed (mean 0 variance 1) weights piped into an activation function do not result in mean 0 variance 1 outputs.
- A standard initialised network starts from a point of saturation.
- Terminology: $\mathrm{fan}_{\mathrm{avg}} = \frac{1}{2}(\mathrm{fan}_{\mathrm{in}} + \mathrm{fan}_{\mathrm{in}})$, where $\mathrm{fan}_{\mathrm{in}} \equiv$ circuit terminology to describe the number of inputs to a layer

Glorot

Activation: None, Tanh, logistic, softmax.
initialise weights: Normal with variance
$\sigma^2 = \frac{1}{\mathrm{fan}_{\mathrm{avg}}}$ (or Uniform in $\pm\sqrt{3/\mathrm{fan}_{\mathrm{avg}}}$)

He

Activation: ReLU *et al.*
initialise weights:
variance $\sigma^2 = \frac{2}{\mathrm{fan}_{\mathrm{in}}}$

Lecun

Activation: SELU
initialise weights:
variance $\sigma^2 = \frac{1}{\mathrm{fan}_{\mathrm{in}}}$

Vanishing/Exploding gradients: Nonsaturating activation functions

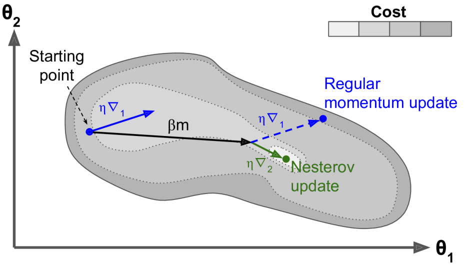
- Another way to fix the saturation problem is to choose a nonsaturating activation function (N.B. This is not how mother nature does it).
- ReLU$=\max(0,z)$ does not saturate, although it has a vanishing gradient by definition for negative inputs.
- LeakyReLU$_\alpha = \max(\alpha z, z)$ solves this. α can be viewed as a hyperparameter (0.2, rand large leak or 0.01 for a small leak), randomised during training, or even fit for along the other parameters.
- ELU (exponential linear unit) is another choice
- Finally SELU are self-normalising ELUs. Very modern.
- GELU also trendy

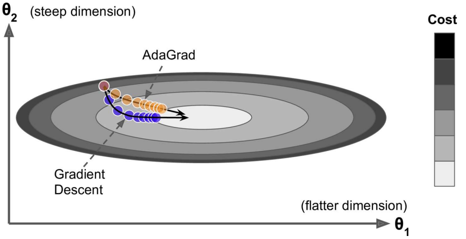
Vanishing/Exploding gradients: Batch normalisation

- Insert a normalisation step (zero centering and normalizing each input using a minibatch)
- At testing we use an exponential moving average over training for the shift parameters.
- Removes the need for standardisation if first layer is a BN layer
- Very much state-of-the-art
- Can be slower than ELU + He

Faster optimisers

- Momentum optimization
  - Give some momentum/gradient memory to the trajectory
- Nesterov Accelerate Gradient
  - Use gradients ahead of the path to nudge
- AdaGrad
  - Scales gradients along steepest directions (measured by accumulated variance in components of gradient)
  - Often stops too early due to agressive downscaling
- RMSProp
  - changes accumulation to an exponential moving average
- Adam & Nadam
  - Adaptive moment estimation
  - Combination of Momentum and RMSProp





Learning rate scheduling

- Power scheduling: $\eta(t) = \eta_0/(1+t/k)^c$
- Exponential scheduling: $\eta(t) = \eta_0 0.1^{t/s}$
- Piecewise constant scheduling
- Performance scheduling: continuous version of early stopping

Regularisation

- ℓ_1 and ℓ_2 regularisation
  - These explicitly constrain the weights, and are easy to apply e.g.

    ```
    layer = keras.layers.Dense(100, activation="elu",
                               kernel_initializer="he_normal",
                               kernel_regularizer=keras.regularizers.l2(0.01))
    ```

    (try it out! `l1` and `l1_l2` as well)
- Dropout
  - At every training step, every neuron is dropped/zeroed in the calculation with probability p
  - No dropping after training (but minor correction of $1-p$ keep probability rescaling)
  - Simple but shockingly effective
- MCDropout
  - Cambridge-based theory paper linking dropout networks with ABC [arxiv:1506.02142]
  - Puts monte-carlo dropping back into training to get errors
- Max-Norm regularisation
  - Constrain the weights so $||w||_2 < r$, clipping if needed $w \to wr/||w||_2$

DeepNet guidelines (Géron)

Hyperparameter	Default value
Kernel initializer:	LeCun initialization
Activation function:	SELU
Normalization:	None (self-normalization)
Regularization:	Early stopping
Optimizer:	Nadam
Learning rate schedule:	Performance scheduling

Don't forget to standardize the input features!

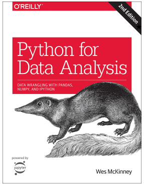
- If you need sparsity try $\ell_1\pm$ FTRL optimisation + BN
- If you need low-latency, use fewer layers, avoid BN, SELU→ReLU, consider sparsity & reducing precision
- If you are risk-sensitive consider MCDropout for performance boost and uncertainty

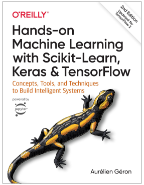
- If self-normalising & overfitting
  add α-dropout
  (do not use other regularisation)
- If cannot self normalise
  try ELU instead of SELU
  (change initialisation)
  use BN after every hidden layer
  try max-norm or ℓ_2 regularisation

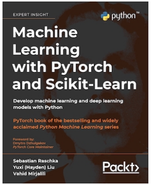
Happy training!

Summary

- The data scientist's Python stack
  numpy, scipy, pandas, matplotlib, sklearn, +Keras/TensorFlow/PyTorch
- Principles & challenges of machine learning
- Theory of training, validation & testing
- Gradient descent and its pitfalls
- Regularisation and early stopping
- Neural networks Anatomy & training
- Recent advances in deep learning
  initialisation, activation, normalisation, optimisation, regularisation




O'REILLY
2nd Edition Updated for TensorFlow 2
Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow
Concepts, Tools, and Techniques to Build Intelligent Systems
powered by jupyter
Aurélien Géron


EXPERT INSIGHT
python
Machine Learning with PyTorch and Scikit-Learn
Develop machine learning and deep learning models with Python
PyTorch book of the bestselling and widely acclaimed Python Machine Learning series
Foreword by: Dmytro Dzhulgakov PyTorch Core Maintainer
Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili
Packt

What we didn't cover!

- Choosing features
- Visualisation
- Classification