**DATA PREP** –

SAMPLING/TEST SET-

>Avoid sampling bias

>Train\_test\_split

>StratifiedShuffleSplit

PIPELINE

>Transformations

>Imputer

>Split cat/numerical data

>Categorical features -

>1hot

>Ordinal

>Integer

>Corr coefs / Corr Matrix

>Scatter plot / matrix

>Feature Engineering

>Feature scaling -

>Normalize

>Standardize

**MODELS**

REGRESSION

Performance measure -

>RMSE

>MAE

LinearRegression -

DecisionTreeRegressor – similar to Classification

RandomForestRegressor – similar to Classification

ExtraTreesRegressor – similar to Classification

SVR –

>sensitive to feature scales (need to scale using StandardScaler, etc)

>can use LinearSVR, or kernel methods

SGDRegressor –

>warm\_start=True allows fit method to continue training where it left off

LogisticRegression –

>uses sigmoid function to output number between 0 and 1

>used as classifier

REGULARIZATION REGRESSION MODELS(data needs to be scaled first!)

Ridge –

>good default

>uses ½ the square of l2 norm

>alpha controls regularization

>SGDRegressor with penalty set to l2 is also ridge

Lasso –

>use when only few features as suspected as useful

>uses l1 norm

>alpha controls regularization

>eliminates weights of least important features(automatic feature selection)

>gradually reduce learning rate to avoid bouncing around optimum

ElasticNet –

>use when only few features as suspected as useful

>mixture of Ridge and Lasso

>can control mix of Ridge vs Lasso by l1\_ratio

CLASSIFICATION

OneVsOneClassifier -

>Multi-class use

>preferred when algos scale poorly like SVC

OneVsRestClassifier

>Multi-class use

>preferred for most binary classification algos

KNeighborsClassifier -

>Can perform both Multilabel and Multioutput

SGDClassifier -

>supports multi-class

>decision function

>threshold

LogisticRegression –

>uses sigmoid function to output number between 0 and 1

>can be regularized by using l1 or l2 penalties

>C controls regularization strength

Softmax Regression –

>Logistic Regression for multi-class

>set LogisticRegression classifier multi\_class="multinomial" and select solver

SVC –

>scales poorly with training data size

>uses OvO automatically for multi-class

>sensitive to feature scales (need to scale using StandardScaler, Normalizer, etc)

>soft vs hard (soft is usually better - regularize by reducing C)

>when using LinearSVC use StandardScaler, set loss=’hinge’, set dual=False

>SVC poly kernel – adjust coef0 to control how much high deg poly effect model

> LinearSVC is faster than SVC(kernel=’linear’)

> SVC Kernels – try linear first, try Gaussian RBF, then experiment with others

DecisionTreeClassifier –

>can be used for multioutput tasks

>doesn’t require much data prep (doesn’t even require feature scaling or centering)

>uses CART algo and therefore produces binary trees - ID3 algo

Can produce trees with more than two children

>needs to be restricted(regularized) – max\_depth, min\_sampls\_split,

min\_samples\_leaf, min\_weight\_fraction\_leaf, max\_leaf\_nodes,

max\_features(increase min\_ feature or reducing max\_ features to regularize)

CLASSIFICATION - ENSEMBLE

RandomForestClassifier -

>supports multi-class

>can use to find feature importance

>has almost all hyperparameters of DecisionTreeClassifier and BaggingClassifier

ExtraTreesClassifier -

>randomizes decision trees more for ensemble

>same API as RandomForestClassifier

BaggingClassifier –

>bagging – sampling with replacement / pasting – sampling without replacement

>bagging oftentimes performs better

>switch to pasting by setting bootstrapping=False

>automatically performs soft voting if classifier has predict\_proba

>oob evaluation

>Create Random Subspaces/Random Patches by adjusting hyperparameters

(bootstrap, bootstrap\_features, max\_samples, max\_features)

VotingClassifier –

>only works when predictors are as independent from one another as possible

>soft VS hard

>soft usually performs better but estimators must have predict\_proba method  
 >to change to soft change voting hyperparameter=’soft’(all classifiers need to have

predict\_probab method for soft voting (for svm change probability=’True’)

AdaBoostClassifier –

>sequential learning

>increases weights for misclassified cases along the way

>use algorithm=’SAMME.R’ when predictors have predic\_proba method (usually

performs better) - sklearn uses ‘SAMME’ otherwise

>if AdaBoost is overfitting, reduce estimators or more strongly regularize base est.

GradientBoostingClassifier –

>sequential learning

>fits every new predictor to residual errors of previous predictor

> to regularize set learning\_rate hyperparameter low (0.1) but will need more

trees to fit model

>use early stopping to find optimal number of trees

>set subsample hyperparameter <1 to implement Stochastic Gradient Boosting

>is possible to use with other cost functions

XGBoostClassifier –

>an optimized library

>has easy early stopping ability

Stacking/Blending –

>Train a model to replace trivial functions like ‘hard voting’

>can be implemented multiple times creating multiple layers

DIMENSIONALITY REDUCTION

>Manifold Learning

>PCA

- set svd\_solver=’randomized’ for quick approx. or ‘full’ for for full SVD

>Randomized PCA

>Incremental PCA

>kPCA

-use Grid Search to find best kernel and gamma hyperparameters

-reconstruction technique (set fit\_inverse\_transform=True) then minimize mse

through grid search

>PCA for Compression/Decompression

-use inverse\_transform to Decompress

>LLE

-manifold technique

>Other popular dimensionality reduction techniques

-Random Projections

-Multidimensional Scaling (MDS)

-Isomap

-t-Distributed Stochastic Neighbor Embedding (t-SNE)

-Linear Discriminant Analysis (LDA)

UNSUPERVISED LEARNING

Clustering –

>used for data analysis, image segmentation, dimensionality reduction, anomaly

detection, semi-supervised learning, preprocessing data

KMeans (Clustering) –

>important to scale data before

>generally one of fastest clustering algorithms

>Hard vs Soft (predict vs transform methods)

>soft clustering can be used for nonlinear dimensionality reduction

>doesn’t always converge to optimal solution – depends on centroid initialization

>MiniBatchKMeans used to speed up and for datasets that don’t fit into memory

>can set initialization location using init hyperparameter

>evaluate using graph of inertia and graph of silhouette score/silhouette diagram

>doesn’t behave well when clusters have varyings sizes, densities, or nonspherical

DBSCAN (Clustering) –

>density based algo (defines clusters as continuous regions of high density)

>doesn’t have predict method so user can choose which classifier to use

>might have trouble if density varies significantly

>also try Hierarchical DBSCAN (HDBSCAN)

Other Clustering Algos–

>Agglomerative clustering

-needs connectivity matrix to scale well (sklearn.neighbors.kneighbors\_graph())

>BIRCH

-designed for large datasets but num of features shouldn’t be to large (<20)

>Mean-Shift

-similar to DBSCAN

-not suited for large datasets

>Affinity propagation

-uses voting system

-not suited for large datasets

>Spectral Clustering

GaussianMixtures –

>needs to be run several times to make sure it converges correctly (like k-means)

>hard/soft ability and generative (can sample new instances)

>set covariance\_type to ‘spherical’, ‘diag’, ‘tied’, or ‘full’ (‘full’ is default, others will

limit range of shapes)

>use for anomaly detection

>use bic/aic to find best k

Other Algos for Anomaly/Novelty Detection

>PCA (using inverse\_transform then comparing reconstruction errors)

>Fast-MCD (using EliipticEnvelope class)

>Isolation Forest

>Local Outlier Factor (LOF)

>One-Class SVM

**EVALUATION**

REGRESSION

RandomForestRegressor - feature\_importances

Performance measures -

>RMSE

>MAE

CLASSIFICATION

Confusion Matrix

Precision/recall

F1 score

ROC curve / PR curve -

>PR curve (when pos class is rare or care more about FP than FN)

>Roc curve (otherwise use Roc curve ^)

>AUC (for ROC curve)

BOTH

Cross validation (expects utility function, not loss function) -

>Cross-val-score

>Cross-val-predict

>StratifiedKFold

Grid Search -

>GridSearchCV

>RandomizedSearchCV

Overfit model (high variance) -

>less complex model / reduce features / increase constraints

>more training data

>reduce noise (errors / outliers)

Underfit (high bias) -

>more powerful model

>better features

>reduce constraints

Plot learning curves

Regularization

Early stopping

Feature importance using RandomForests

**NEURAL NETWORKS**

Perceptron -

>single layer of fully connected neurons

>equivalent to SGDClassifier with loss=’perceptron’

>doesn’t output probability – uses step function as activation function

>can only solve linearly separable problems like logistic classifiers

MLP –

>composed of one or more layers of TLU’s

>popular activation functions

>logistic (sigmoid)

>tanh

>ReLU

>Regression Architectures

>input neurons (one per feature)

>hidden layers (typically 1 to 5)

>neurons per hidden layer (typically 10 to 100)

>output neurons (1 per prediction dimension)

>Hidden activation (ReLU or SELU)

>Output activation (none, or ReLU/softmax for pos. out, or

logistic/tanh for bounded)

>Loss function (MSE or MAE/Huber (if outliers)

>Classification Architectures

>input neurons (one per feature)

>hidden layers (typically 1 to 5)

>neurons per hidden layer (typically 10 to 100)

>output neurons (1 for binary, 1 per class for Multiclass)

>Hidden activation (ReLU/tanh/logistic(sigmoid))

>Output activation (logistic for binary, softmax for Multiclass)

>Loss function (cross entropy)

>set kernel\_initializer/bias\_initializer hyperparameter to change weight and bias

initialization method

>use loss= ‘sparse\_categorical\_crossentropy’ for class labels,

loss=’categorical\_crossentropy’ for one-hot vectors, and

loss = ‘binary\_crossentropy’ for binary labels (for binary set act func to sigmoid)

>to convert sparse labels (class indices) to one\_hot vector labels, use

keras.utils.to\_categorical(). To do opposite use np.argmax() with axis=1

>use validation\_split instead of validation\_data to have keras split for you

>can also adjust hyperparameters class\_weight for underrepresented

classes and sample\_weight for per\_instance weights

>when plotting training curve, it should be shifted by half and epoch to left

>use model.save\_weights and model.load\_weights for subclassing api

Tensorboard – NEEDS WORK!

Fine-Tuning –

>GridSearch and RandomGridSearch technique

>Other opt libraries – Hyperopt, Hyperas/kopt/Talos(for keras models), Keras Tuner

Scikit-Optimize, Spearmint, Hyperband, Sklearn-Deap

Hyperparameters –

>Layers (increasing layers vs increasing neurons per layer gives more bang for buck)

>Neurons

>Learning rate

- potentially the most important hyperparameter

- needs to be adjusted when any other hyperparameter is changed

- can find good learning rate by gradually increasing for each interation, then graph

>Optimizer

>Batch size

- 2 to 32

>Activation function

- ReLU is good default

>Number of interations

- just use early stopping

TensorFlow Playground!

**TRAINING DEEP NEURAL NETWORKS**

VANISHING/EXPLODING GRADIENT PROBLEM

Initializers -

>pairing correct Initialization with activation functions

- Initialization = Glorot Activation = None, tanh, logistic, softmax

- Initialization = He Activation = ReLU and ReLU variants

- Initialization = LeCun Activation = SELU

>Keras default is Glorot with uniform distribution

>can use normal or uniform

Activation Functions (Nonsaturating) –

>ReLU - use with He initialization

- can suffer from dying ReLUs (some neurons die off and only output 0)

>LeakyReLU - use with He initialization

- neurons won’t die off

- usually outperforms ReLU

- set alpha to create diff leak sizes (0.2 is big leak but worked well)

>RReLU (randomized leaky ReLU) - use with He initialization

-alpha is picked randomly during training and fixed to ave during testing

-acts like regularizer

-performs fairly well

>PReLU (parametric leaky ReLU) - use with He initialization

-alpha is learned during backpropagation

-might strongly outperform ReLU on large image datasets, but risks overfitting smaller

>ELU (exponential linear unit) - use with He initialization

-shown to outperform all ReLU variants

-slower to compute but converges faster to make up for it

>SELU – use with LeCun initialization

-will self-normalize if:

-network is built exclusively of dense layers

-all hidden layers have SELU act func

-input features are standardized (mean - 0, SD - 1)

-every hidden layers weights initialized with ‘lecun\_normal’

-network architecture must be sequential

-will outperform other networks that meet same criteria

>In General – SELU > ELU > LeakyReLU(and variants) > ReLU > tanh > logistic  
 -but depends on circumstances

>use keras.activations and keras.layers to access different activations

Batch Normalization –

>add before or after activation function in each layer

>if added before first layer, it will approximately standardize your inputs for you

>adds 4 new parameters to be learned during training

>reduces vanishing grad problem to the point that saturating activation functions can

possibly be used

>reduces network sensitivity to weight initialization

>provides the use of larger learning rates

>acts like regularizer, reducing need for other techniques

>slows down training but speeds up convergence

>if adding before act func, remove act func from hidden layers and add them as

separate layers while removing bias term from prev layer by set use\_bias=False

>can adjust momentum hyperparameter to something close to 1 (0.9, 0.99, 0.999)

>adjust axis hyperparameter if input batch is 2D or multiple dimensions

>BN has become extremely used and almost a given in most networks

Gradient Clipping –

>most often used in RNN’s where BN is tricky

>set it when creating an optimizer

>can set using clipvalue or clipnorm

REUSING PRETRAINED LAYERS

Transfer learning –

>first try freezing all reused layers, then move on to unfreezing top layers with

reduced learning rate. If performance still lacks, drop top layers and iterate

through process again until correct num of layers is found

>must always compile model after freezing/unfreezing layers

>use clone\_model to make copy of original model so it isn’t affected by training,

then use get\_weights and set\_weights on cloned model

Unsupervised pretraining –

>used to use RBMs, now typically use autoencoders or GANs

Pretraining on Auxiliary task –

>find related task to train model, then use transfer learning

Self-supervised learning –

>automatically generate labels from data itself

FASTER OPTIMIZERS

Momentum optimization –

>faster than gradient descent but tends to overshoot and oscillate before stabilizing

>adds momentum vector and friction hyperparameter

>usually works well with momentum=0.9

Nesterov Accelerated Gradient (NAG) -

>almost always faster than vanilla momentum optimizer and converges faster

>similar to vanilla momentum but measures gradient of cost function slightly ahead

>set neterov=True and can set momentum=0.9

AdaGrad –

>don’t use because it doesn’t converge

>requires less tuning of learning rate but slows down too fast to converge

>heads towards optimum earlier instead of going straight down steepest slope

RMSProp –

>fixes early stopping problem with AdaGrad

>setting decay rate hyperparameter rho=0.9 usually works well

>almost always outperforms AdaGrad and was preferred opt algo until Adam came

Adam (adaptive moment estimation) –

>combines ideas of momentum opt and RMSProp

>requires less tuning of lr since it is adaptive like AdaGrad and RMSProp

>3 hyperparameters beta1= 0.9, beta2=0.999 and epsilon=10e-7

AdaMax (Adam variant) –

>can be more stable than Adam, but Adam still usually performs better

Nadam (Adam variant) –

>is Adam optimizer with Nesterov trick

>converges faster than Adam

Adaptive Optimizers –

>RMSProp, Adam, Nadam are often great and converge fast to a good solution

>sometimes generalize poorly so try NAG when this is the case

Optimizer Overview –

>SGD Conv spd = \* Conv Qual = \*\*\*

>SGD (momentum=) Conv spd = \*\* Conv Qual = \*\*\*

>SGD (momentum= , nesterov=True) Conv spd = \*\* Conv Qual = \*\*\*

>Adagrad Conv spd = \*\*\* Conv Qual = \* (stops early)

>RMSProp Conv spd = \*\*\* Conv Qual = \*\* or \*\*\*

>Adam Conv spd = \*\*\* Conv Qual = \*\* or \*\*\*

>Nadam Conv spd = \*\*\* Conv Qual = \*\* or \*\*\*

>AdaMax Conv spd = \*\*\* Conv Qual = \*\* or \*\*\*

Learning Rate Scheduling –

>Power scheduling

-set learning rate to function of iteration number

-drops quickly at first then more and more slowly

>Exponential scheduling

-learning rate gradually drops by power of 10

>Piecewise constant scheduling

-using constant lr for number of epochs and continually dropping lr

>Performance scheduling

-measure val error like w/ early stopping, and reduce lr when error stops dropping

>1cycle scheduling

-starts by increasing initial lr n0 to n1 for first half of training, then drops back to

n0 for second half of training, then finishes off by decreasing lr several orders

of magnitude for last few epochs

-when used with momentum can speed up training and provide better performance

-when used with momentum start with high momentum (0.95), then drop to

lower linearly during first half (0.85), then bring back to maximum during second

half (0.95)

>both performance and exponential lr scheduling with momentum optimization showed

to perform well

>when using lr argument, training can start where it left off if loading a saved model

because optimizer and lr gets saved with model. This is not the case when using

epoch argument.

REGULARIZATION

L1 and L2

>use keras.regularizers to implement

>L2 to constrain weights

>L1 for a sparse model (many weights equal to 0)

>L1 and L2

Dropout

>neurons are temporarily dropped out with specified prob (incl inputs, excl outputs)

>rate is typically set between 0.1-0.5 : 0.2-0.3 for RNNs and 0.4-0.5 for CNNs

>networks tend to generalize better

>usually apply dropout to last 1-3 layers excl output (sometimes just last layer)

>comparing training loss vs val loss can be misleading while using dropout. Model

may be overfitting but have similar training/val loss. Evaluate training loss without

dropout (after training) to get accurate measure.

Alpha Dropout

>if network is self-normalizing using SELU activation, use alpha dropout to preserve

mean and SD of inputs

Monte Carlo Dropout

>boosts model performance of any trained dropout model

>provides excellent measure of uncertainty regarding predictions

>need to create class if model has layers that behave in special way (BatchNormal…)

Max-Norm Regularization

>reducing hyperparameter ‘r’ increases regularization

>can also help alleviate unstable gradients

>doesn’t add regularization loss term but rescales weight vector if needed

DEFAULT CONFIGS

Default DNN config –

Kernel Initializer He initialization

Activation Function ELU

Normalization None if shallow;Batch Norm if deep

Regularization Early stopping (+L2 reg. if needed)

Optimer Momentum optimization (or RMSProp or Nadam)

Learning rate schedule 1cycle

Default DNN config (self-normalizing) –

Kernel Initializer LeCun initialization

Activation Function SELU

Normalization None (self-normalization)

Regularization Alpha dropout if needed

Optimer Momentum optimization (or RMSProp or Nadam)

Learning rate schedule 1cycle

**Lower level TensorFlow API**

Customize everything and work with TensorFlow operations

- Chapter 12

- other data types - strings, string arrays, ragged tendsors, sparse tensors, sets, tensor

arrays, etc.

- customizable - loss funcs, act funcs, initializers, regularizers, constraints, metrics,

layers, models, training loops, etc.

- <https://github.com/ageron/handson->

ml2/blob/master/12\_custom\_models\_and\_training\_with\_tensorflow.ipynb

**Data API and Preprocessing**

- Data API vs Preprocessing/Custom Preprocessing Layers vs TF Transform (TFT)

Possibly -

> If the step is required at both training and prediction time -> Preprocessing layer

> If the step is required only during training -> No Preprocessing layer, use Dataset

Operation

Dataset API

- build preprocessing function

- create train, val, test set by plugging data into function

- use prefetch to have each batch prepared one step ahead of fitting

- can be used to preprocess large data supply gradually one batch at a time

Categorical data (encoding vs embedding)

- categories < 10, one-hot encoding is generally good

- categories > 50, embedding is generally better

- categories between 10 - 50, experiment with both one-hot and encoding

- embedding is equivalent to one-hot encoding followed by dense layer

with no act func and no biases

**Convolution Neural Networks (CNN)**

Convolution layer

- don’t use too large of kernel size (instead of 5x5 use two 3x3 layers)

- exception can be made for first layer (can have large kernel – 5x5 usually w/

stride of 2 or more)

Pooling layer

- reduces computations/memory usage/num parameters AND introduces invariance

to small translations

- max pooling (used more often), ave pooling, global ave pooling

- depth-wise pooling - CNN can learn any invariance

Architectures

> LeNet-5

- widely known and widely used for handwritten digit recognition (MNIST)

> AlexNet

- top-five error rate of 17% (2012 ImageNet ILSVRC challenge)

- similar to LeNet-5 but much larger and deeper

> GoogLeNet

- top-five error rate of < 7% (2014 ILSVRC challenge)

- much deeper using inception modules

> VGGNet

- runner-up next to GoogLeNet (2014 ILSVRC challenge)

- simple and classical architecture w/ 2-3 conv layers then a pool layer and so on..

> ResNet

- top-five error rate of < 3.6% (2015 ILSVRC challenge)

- used extremely deep CNN with 152 layers

>Inception-v4

- merged ideas of GoogLeNet and ResNet

- top-five error rate near 3% on ImageNet classification

> Xception

- variant of GoogLeNet and ResNet

- replaced inception modules with layer called depthwise separable conv layer

- consider using depthwise separable conv layers as default except after layers with

few channels

> SENet (Squeeze-and-Excitation Network)

- top-five error rate of 2.25% (2017 ILSVRC challenge)

- extends existing architectures such as inception nets and ResNets and boosts

Performance by adding small neural net called and SE block to every unit

- when applied to Inception and ResNets they are then called SE-Inception and

SE-ResNet

**Sequences with RNNs/CNNs**

Input shape

- [batch size, time steps, dimensionality]

MC Dropout

- add MC Dropout layer within each memory cell dropping part of inputs/hidden states

Unstable Gradients problem

- nonsaturating act function (such as ReLU) may not help

- reduce learning rate or simply use a saturating act func like hyperbolic tangent

- can monitor gradients using TensorBoard and maybe consider Gradient Clipping

> Batch Normalization

- only slightly beneficial

- apply to inputs, not to hidden states (between rec layers, not within rec layers)

>Layer Normalization

- often works better than Batch Normalization

- normalizes across features dimension instead of batch dimension

- typically used right after linear combination of inputs and hidden states

- is performed after linear comb but before act func

>Dropout

- all recurrent layers except RNN have dropout/recurrent\_dropout hyperparameters

- dropout for dropout rate of inputs at each time step

- recurrent\_dropout for dropout rate of hidden states at each time step

LSTM

- help with short term memory problem

- peephole connection can be added by passing PeepholeLSTM cell to layers.RNN

GRU

- simplified LSTM but shown to perform just as well

Conv1D

WaveNet

- stacks Conv1D layers while changing dilation\_rate