* (The **learning rate decayed** by a factor of 1.0000002 every batch update until it reached a minimum of 10^−6,)
* (An additional boost in performance is obtained by using the **dropout** training algorithm, in which we stochastically drop neurons in the top hidden layer with 50% probability during training.)
* (Done) We selected a five-layer neural network with 300 hidden units in each layer,
* (Done) a **learning rate** of 0.05, and a **weight decay** coefficient of 1 × 10−5.
* (Done)Hidden layers have **tanh activation function**
* (Done)Gradient computations were made on **mini-batches** of size 100
* An additional boost in performance is obtained by using the dropout training algorithm, in which we stochastically drop neurons in the top hidden layer with 50% probability during training.
* **Weights were initialized** from a normal distribution with zero mean and standard deviation, 0.1 in the first layer, 0.001 in the output layer, and 0.05 all other hidden layers.
* A **momentum** term increased linearly over the first 200 epochs from 0.9 to 0.99, at which point it remained constant.
* The **learning rate decayed** by a factor of 1.0000002 every batch update until it reached a minimum of 10^−6,

PyLearn2 -

Weight Initialization – do not make all initialization 0,

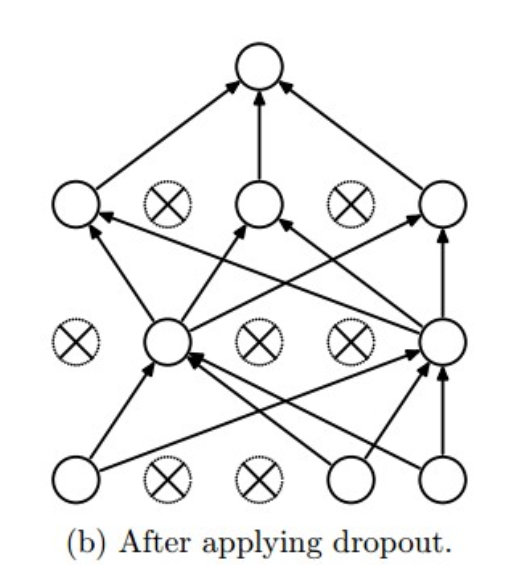
Learning Rate – controls quickly/slowly NN learns a problem, configurable hyperparameter, usually small positive value range 0.0 and 1.0, controls apportioned error that weights of model are updated with each time they are updated (i.e. at the end of batch training), large learning rate – model learns faster at cost of arriving on sub-optimal final set of weights/smaller learning rate may allow better model to learn optimal but longer to train; traditional default 0.1 or 0.01; scalar used to train model via gradient descent; during each iteration, gradient descent algorithm multiplies learning rate by gradient – result is gradient step. Controls how much to change the weight to correct for error (i.e. value of 0.1 will update the weight 10% of amount it could be updated), small rates preferred that cause slower learning over larger number of iterations; amount of change to model during each step of searching for optimal solution (global vs local), (aka step size), positive scalar determining the size of step, learning rate less than 1.0 and greater than 10^-6

Smaller learning rates require more training epochs, larger rates have more rapid change

Smaller batch sizes better suited to smaller learning rates

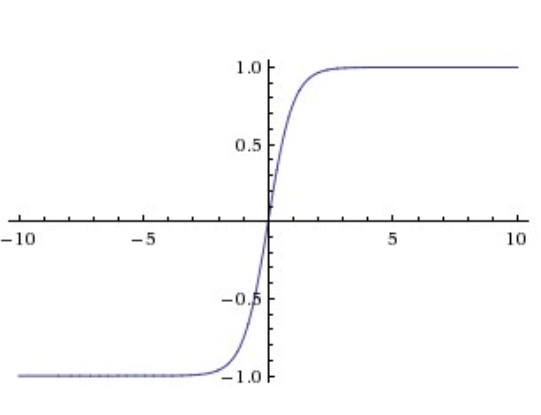
Learning Rate Decay – the way in which learning rate changes over time / training epochs, decrease learning rate linearly from large initial value to small value, large weight changes at beginning of process, small/fine-tuning toward the end, may be best practice now; can decay to a small value close to zero, or decay over fixed number of training epochs and kept constant at small value

Drop Out – form of regularization, removes a random selection of fixed number of units in a network layer for a single gradient step; more dropped, the stronger the regularization; implemented by only keeping a neuron active with some probability p, or setting to 0 otherwise; roughly doubles the number of iterations required for convergence but training time for each epoch is less



Weight Decay -

Activation Functions/Transfer Functions – takes weighted sum of all inputs from previous layer and generates/passes an output value to next layer; transforms summed weighted input from node into the activation of node or output for that input; for node, defines output of node given input or set of inputs

* Tanh activation – cannot be used in networks with many layers due vanishing gradients, nonlinear activation function, outputs values between -1.0 and 1.0, can have limited sensitivity and saturation, output is 0-centered
* 
* Sigmoid activation – S shape, logistic function, take any input and produce result between 0 and 1, cannot be used with many layers due to vanishing gradients, nonlinear activation function, large negatives become 0/large positives become 1, drawbacks – sigmoids saturate, kill gradients; sigmoid outputs are not 0-centered,
* Softmax activation –
* ReLU activation – rectified linear activation function, piecewise linear function that outputs the input directly if positive (otherwise, output 0). Default activation, easy train, better performance, activation is threshold at 0, can accelerate SGD, implemented by simply thresholding matrix of activations at 0, can be fragile where weights could update in a way for neuron to not activate again (Leaky ReLU attempts to fix dying problem)

Backpropagation Algorithm – method for training weights, supervised learning method for multilayer feed-forward networks, calculate error for each output neuron to get error signal (input) to propagate backward through network

Momentum – does not make it easier to configure learning rate, step size is independent of momentum, improve speed of optimization in concert with step size value greater 0.0 and less than 1.0, common values 0.9, 0.99, 0.5; change to stochastic gradient descent when exponentially weighted average of prior updates to weight can be included when weights are updated, adds inertia to update procedure to continue to move in one direction, accelerate learning especially with high curvature, small/consistent gradients, noisy gradients .. aka velocity, smoothing optimization process

Batch – set of examples used in one iteration (one gradient update) of model training, controls accuracy of estimate of error gradient when training, (3 types – batch, stochastic, minibatch gradient descent), larger batch == more training examples == more accurate estimate, hyppar controls number of training samples to work before internal parameters are updated

Batch size – number of examples in batch, batch size of SGD is 1; batch size of mini-batch is usually between 10 and 1000; number of examples from training dataset used to estimate the error gradient; impacts how quickly model will learn and stability of learning process

i.e. Batch Size of 32 means 32 samples from training dataset will be used to estimate error gradient before model weights are updated (32 is a good default)

One training epoch mean learning algorithm has made one pass through training dataset, where examples were separated into randomly selected batch size groups

Batch size and number of batches are different

Divide dataset into Number of Batches (sets, parts)

Iterations is the number of batches needed to complete on epoch

* **Batch Gradient Descent**. Batch size is set to the total number of examples in the training dataset.
* **Stochastic Gradient Descent**. Batch size is set to one.
* **Minibatch Gradient Descent**. Batch size is set to more than one and less than the total number of examples in the training dataset.

Epoch – a full training pass over the entire dataset such that each example has been seen once; epoch represents N/batch size training iterations, where N is the total number of examples. Loop through fix number of epochs and within each, update network for reach row in training data, number of complete passes through training dataset, one epoch mean that each sample in training dataset had opportunity to update internal parameters, epoch has one or more batches. Usually large, allowing learning algo to run until error from model is sufficiently minimized; one epoch is when an entire dataset is passed forward and backward through NN only once

“You can think of a for-loop over the number of epochs where each loop proceeds over the training dataset. Within this for-loop is another nested for-loop that iterates over each batch of samples, where one batch has the specified “batch size” number of samples.”

AUC / ROC CURVE – evaluation metric that considers all possible classification thresholds; areas of ROC curve is probability that a classifier will be more confident that a randomly chosen positive example is actually positive than that a randomly chosen negative example is positive

Neuron – has set of weights, one weight for each input connection and an additional weight for bias.

NN – input layer is row from training set; real first layer is hidden layer, then output layer that has one neuron for each class value

Forward-propagation – calculate output from neural network by propagating an input signal through each layer until the output layer outputs its values, generate predictions during training that will need to be correct and also used to make predictions on new data

Weight Regularization – (large network weights may be indicative of instability, where small changes in input lead to large changes in output…can be sign of overfitting). Method to keep weights small, reduce overfitting, improve model generalization, vector norm of weights is calculated per layer, rather than for whole network, can use both L1 and L2

* Calculate the sum of the absolute values of the weights, called L1. Encourages weights to 0.0, resulting in more sparse weights (weights with more 0.0 values)
* Calculate the sum of the squared values of the weights, called L2. More nuanced, penalizing larger weights more, but results in less sparse weights, more traditional, referred to as “weight decay”

\*\*more layers and more nodes tends to overfit training data

#### Dropout -

Dropout technique works by randomly reducing the number of interconnecting neurons within a neural network. At every training step, each neuron has a chance of being left out, or rather, dropped out of the collated contribution from connected neurons.

This technique minimizes overfitting because each neuron becomes independently sufficient, in the sense that the neurons within the layers learn weight values that are not based on the cooperation of its neighbouring neurons.

#### Momentum -

momentum helps SGD to navigate along the relevant directions and softens the oscillations in the irrelevant. It simply adds a fraction of the direction of the previous step to a current step. This achieves amplification of speed in the correct direction and softens oscillation in wrong directions. This fraction is usually in the (0, 1) range. It also makes sense to use adaptive momentum. In the beginning of learning a big momentum will only hinder your progress, so it makes sense to use something like 0.01 and once all the high gradients disappeared you can use a bigger momentum. There is one problem with momentum: when we are very close to the goal, our momentum in most of the cases is very high and it does not know that it should slow down. This can cause it to miss or oscillate around the minima; momentum term increases for dimensions whose gradients point the same direction / reduces updates for dimensions whose gradients change direction

Naming Convention – single layer NN describes network with no hidden layer (input maps to output) \*\*in case do we need to comment on the number of learnable parameters??\*\*

Case Recommendations –

* 99% train vs 1% test – we would suggest more traditional 80% train vs 20% test
* Explore new algorithms developed

References:

<https://developers.google.com/machine-learning/glossary>

<https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/>

<https://machinelearningmastery.com/what-is-deep-learning/>

<https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/>

<https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/>

<https://en.wikipedia.org/wiki/Activation_function>

<https://machinelearningmastery.com/weight-regularization-to-reduce-overfitting-of-deep-learning-models/>

<https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/>

<https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/>

<https://cs231n.github.io/neural-networks-2/>

<https://arxiv.org/pdf/1206.5533.pdf>

<http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf>

<https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9>

<https://cs231n.github.io/neural-networks-1/#arch>

<https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5>

<https://ruder.io/optimizing-gradient-descent/>

Sample Code

|  |
| --- |
| # mlp for the blobs problem with minibatch gradient descent  from sklearn.datasets import make\_blobs  from keras.layers import Dense  from keras.models import Sequential  from keras.optimizers import SGD  from keras.utils import to\_categorical  from matplotlib import pyplot  # generate 2d classification dataset  X, y = make\_blobs(n\_samples=1000, centers=3, n\_features=2, cluster\_std=2, random\_state=2)  # one hot encode output variable  y = to\_categorical(y)  # split into train and test  n\_train = 500  trainX, testX = X[:n\_train, :], X[n\_train:, :]  trainy, testy = y[:n\_train], y[n\_train:]  # define model  model = Sequential()  model.add(Dense(50, input\_dim=2, activation='relu', kernel\_initializer='he\_uniform'))  model.add(Dense(3, activation='softmax'))  # compile model  opt = SGD(lr=0.01, momentum=0.9)  model.compile(loss='categorical\_crossentropy', optimizer=opt, metrics=['accuracy'])  # fit model  history = model.fit(trainX, trainy, validation\_data=(testX, testy), epochs=200, verbose=0, batch\_size=32)  # evaluate the model  \_, train\_acc = model.evaluate(trainX, trainy, verbose=0)  \_, test\_acc = model.evaluate(testX, testy, verbose=0)  print('Train: %.3f, Test: %.3f' % (train\_acc, test\_acc))  # plot training history  pyplot.plot(history.history['accuracy'], label='train')  pyplot.plot(history.history['val\_accuracy'], label='test')  pyplot.legend()  pyplot.show() |

Dataset with 200 samples

Batch size = 5

Epochs = 1000

* Dataset will be divided into 40 batches, each with 5 samples, model weights will update after each batch of 5 samples
* One epoch will involve 40 batches/40 updates to model
* 1000 epochs, model will be exposed/passed through whole data 1000 times, total of 40,000 batches during entire training process
* The recommended preprocessing is to center the data to have mean of zero, and normalize its scale to [-1, 1] along each feature
* Initialize the weights by drawing them from a gaussian distribution with standard deviation of 2/n−−−√2/n, where nn is the number of inputs to the neuron. E.g. in numpy: w = np.random.randn(n) \* sqrt(2.0/n).