Basic Neural Networks

Ilya Antonov

September 13, 2019

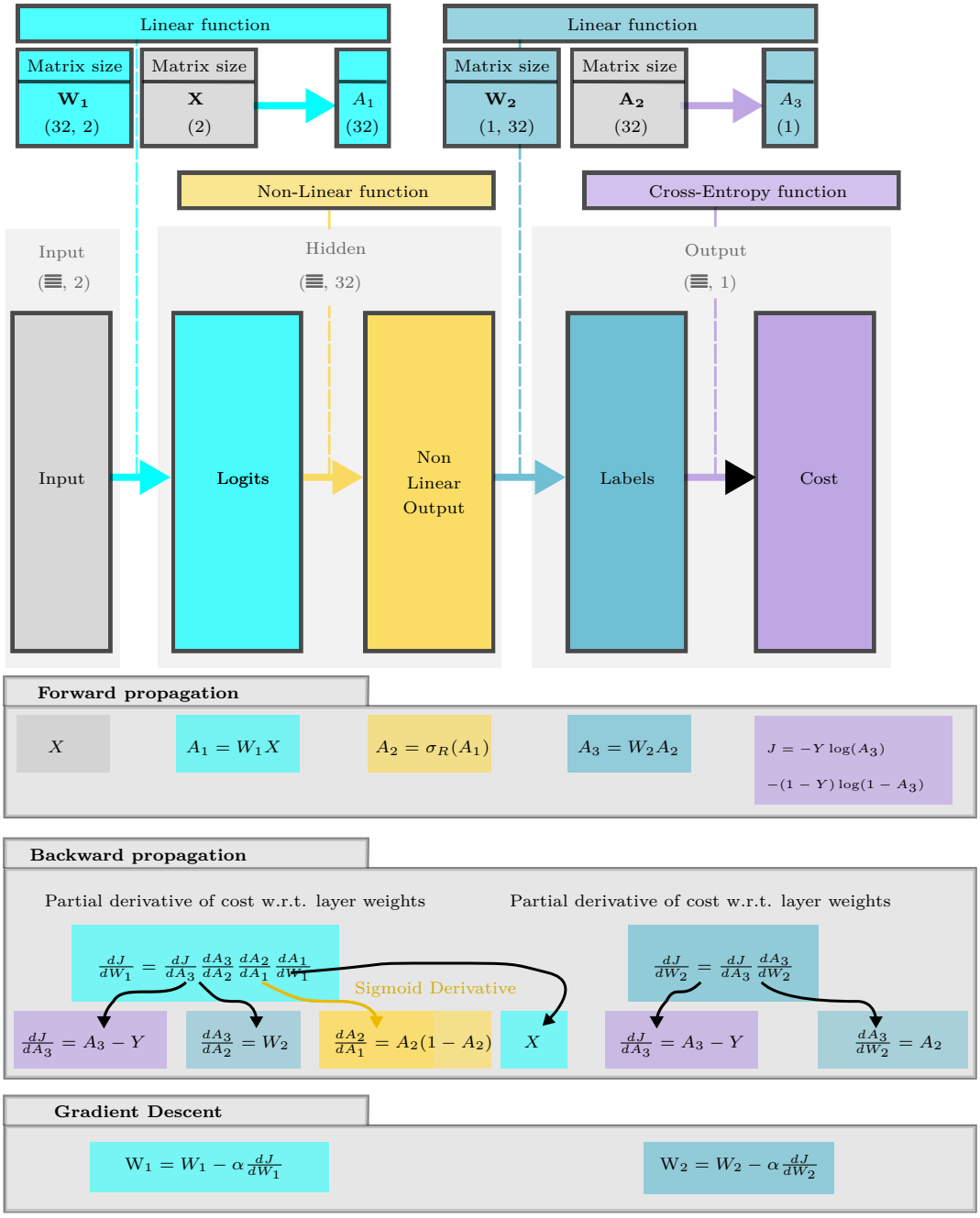
Contents

SECTION 1

Standard Neural Network

The most standard neural network (NN) is demonstrated below. We will quickly cover it's main aspects, and move onto describing ways of:

- Preventing overfitting of NN Sec. 2;
- Initializing weights and input data Sec. 3;
- Processing the data in batches for reactiveness Sec. 5;
- Improving training speed Sec. 6;
- Additional NN layers: `Batch Normalization Layers`, `Softmax Regression`;
- Full review of model parameters Sec. 9;



The NN accepts an input vector of two elements and maps it to a single output value through a hidden layer of 32 nodes. During training the network will need a lot of these vectors (the number of training vectors is denoted by ☰) which are passed in as one big array. The dimensions of these training arrays are (☰, 2) and (☰, 1) correspondingly.

It is important not to forget that that training happens on ☰ input and output vectors, and that these must be supplied as arrays.

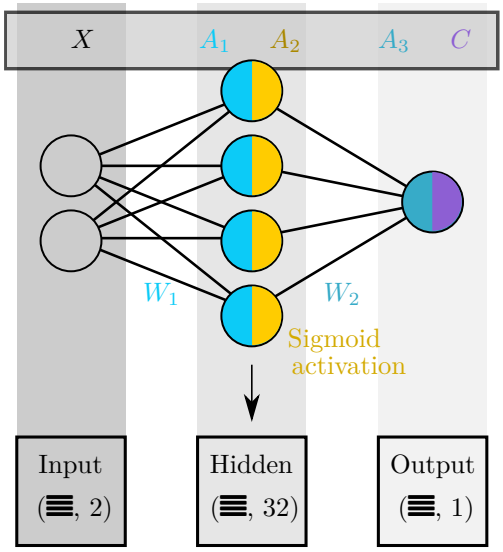


Figure 1: NN processes a input vector `X.dim = 2` into an output value `A3.dim=1`, which is compared to the target label `Y.dim=1`. `A1` and `A2` are intermediate values within the network. As during training this would need to be repeated many times, many such vectors are stacked into (☰, 2) and (☰, 1) arrays.

Every epoch, the NN adjusts weights W_1, W_2 to minimize the `Cost Function`

$$J = -Y \log(A_3) - (1 - Y) \log(1 - A_3),$$

which is simply a metric of how close the evaluated vector, $A3$, is to the desired label Y. Each `epoch` should decrease this cost function, and the network converges on the optimal weights for that particular training set. The 3 import steps in every `epoch` are:

1. **Forward Propagation:** An input X, undergoes a series of transformations, including `Sigmoid Activation`, which will produce:
   - An output A_3;
   - A cost function J, that quantifies how close the output, A_3, is to the target value, Y. A superb read about the meaning of the cost function is given in this blog post.
2. **Backward Propagation:** Evaluation the dependencies of the `Cost Function` (which we are trying to minimize) on the different parameters of the network by taking partial derivatives:

$$\frac{\partial J}{\partial W_1} \qquad \frac{\partial J}{\partial W_2}$$

3. **Gradient Descent:** Using the `learning rate`, α, weights are adjusted by their impact on the `Cost Function` i.e. proportional to the partial derivative:

$$W_1 = W_1 - \alpha \frac{dJ}{dW_1} \qquad W_2 = W_2 - \alpha \frac{dJ}{dW_2}$$

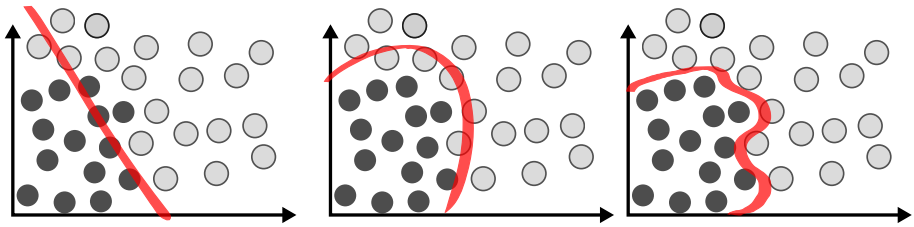
1.1 Example code in keras

```
# X — input array of N elements (each element can itself be an array)
# Y — output array of N elements
# 1 — Input
INPUTS = ks.Input((X.shape[1], ))
# 2 — Hidden layer with sigmoid activation
HIDDEN_LAYER = ks.layers.Dense(32, activation='sigmoid')(INPUTS)
# 3 — Output
OUTPUT_LAYER = ks.layers.Dense(Y.shape[1])(HIDDEN_LAYER)
# 4 — Create model
model = ks.Model(inputs=INPUTS, outputs=OUTPUT_LAYER)
model.compile(loss=ks.losses.categorical_cross_entropy)
```

SECTION 2

REGULARIZATION

There are two measures for the quality of a trained NN. **Bias** measures how well the NN fits the training data; **Variance** measures how well the NN fits the test data.



High Bias Optimal High Variance

High Bias would mean that the network is poorly trained to predict anything. To fix it use:

- Bigger network;
- Different NN architecture;
- More training data.

High variance means that the NN overfits to our training dataset. Generally a NN left to itself would adjust it's weights to perfectly fit the training dataset and will perform poorly if new data, for which it was not optimized, is added. To fix it use:

- `Regularizers`;
- `Dropout`.

2.1 Regularizers

A regularizer penalizes the NN for using large weights by adding an additional term to the cost function:

$$J(\vec{w}, \vec{y}_{nn}, \vec{y}_{\text{label}}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y^i_{\text{label}}, y^i_{nn}) \quad + \quad \begin{cases} \frac{\lambda}{2N}\sum_i ||w_i||^2 & \text{L2 regularization} \\ \frac{\lambda}{2N}\sum_i |w_i| & \text{L1 regularization} \end{cases}$$

- $\mathcal{L}$ is the original cost function that measures the difference between the real values, y_{label}, and NN output, y_{nn}. It could be `cross entropy`, as above, or a simple `means square difference`;
- N number of training samples;
- $\vec{w}$ collection of all the weights used by the NN as a 1D vector.
- λ parameter that sets the regularization strength.

When the cost function is differentiated during backpropagation, this extra term delivers contribution:

$$\frac{dJ}{dw_i} = \left[\frac{dJ}{dw_i}\right]_{\text{original}} + \begin{cases} \frac{\lambda}{N} w_i & \text{L2 regularization} \\ \frac{\lambda}{2N} & \text{L1 regularization} \end{cases}$$

which will be used in the gradient decay correction of the weights (shown here for `L2 regularization`)

$$w_i = w_i - \alpha \frac{dJ}{dw_i} = w_i - \alpha \left[\frac{dJ}{dw_i}\right]_{\text{original}} - \alpha \frac{\lambda}{N} w_i.$$

- **Every `epoch`, the weights undergo an additional suppression as a result of the regularizers.**
- `L2 regularization` is more aggressive than `L1 regularization`.
- λ **controls the strength of regularization**;
- Defining a `Dense layer` in whose weights will be regularized in the `cost function`:

```
lambd = 0.03

layer = ks.layers.Dense(112, kernel_regularizer=ks.regularizers.l1(lambd))
```

With moderated weights, the model will not be able to overfit to the training set. In the extreme case that most of the weights are suppressed to 0, there is effectively no flow of information within the network, and the effective size of the network decreases.

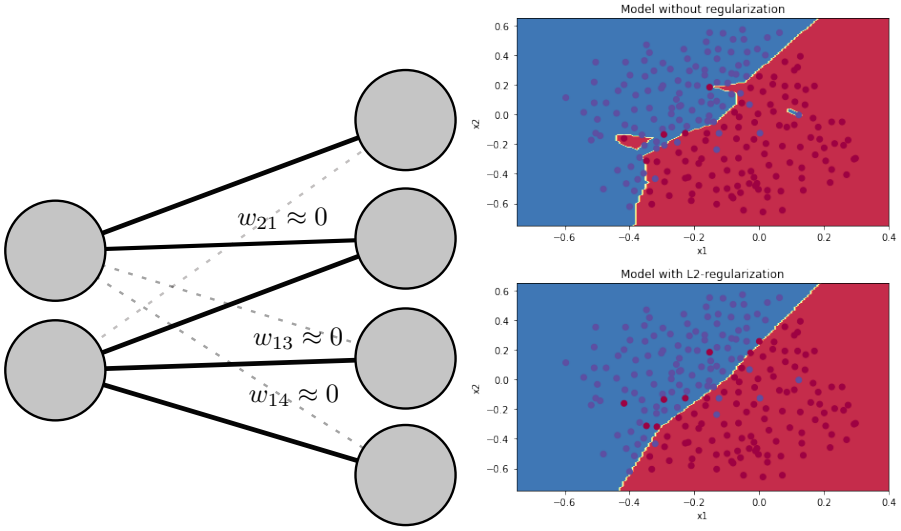


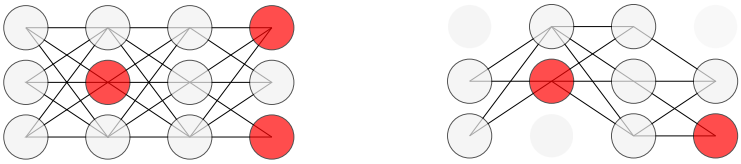
Figure 2: Regularizers may set certain weights to zero, reducing the size of the network. On the left is an example of how `L2 regularization` suppresses some of the weights, making the model simpler, as a result, smoother and less choppy.

2.2 `Dropout`

In dropout, certain weights are set to 0 during each `epoch`. Thus each iteration the NN trains a different model that uses only a subset of the neurons. The model needs to diversify over all the available neurons, and less likely to have preference on a persistent subset.

Without getting reliant on specific features during training and optimizes across all of the weights.

1. To perform dropout, create an array of 1s of the exact same dimensions as the weight array;



Emphasis on red elements Dropout of elements

Figure 3: Setting weights to zero ensures that the NN does not fixate on specific weights. On the right image, the setting the top-right weight to zero forces the network to optimize over the other weights.

2. With a probability p_{dropout} set it's elements to zero;
3. Using this mask, perform an element wise multiplication with the weight array to get the new weight matrix;
4. To avoid the cost function from changing too much, the loss of some weights it made up by scaling the rest by $1/(1-p_{\text{dropout}})$:

$$\begin{cases} W_1 = \begin{pmatrix} 0.2 & 0.4 & 0.3 & 0.4 \\ 0.11 & 0.43 & 14 & 88 \\ 0 & 0.7 & 1 & 2 \end{pmatrix} \\ \text{Dropout} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \xrightarrow[0.25]{p_{\text{dropout}}} \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{cases} \Rightarrow \frac{W_1 \odot \text{Dropout}}{1-p_{\text{dropout}}} = \frac{4}{3} \begin{pmatrix} 0 & 0.4 & 0 & 0.4 \\ 0 & 0.43 & 14 & 0 \\ 0 & 0.7 & 1 & 2 \end{pmatrix}.$$

- Generate a new dropout matrix for each new `epoch`, that randomly shuts down some neurons;
- Update the non-zero weights during gradient descent;
- **Set the probability of dropping out nodes** $0.2 < p_{\text{dropout}} < 0.5$;
- Can also perform dropout on the `Input` and `Output`.
- Defining a `Dropout layer` in `keras`:

```
# 1 - define the dropout rate
rate = 0.2
# 2 - initialize a dropout layer
layer = ks.layers.Dropout(rate)(PREVIOUS_LAYER)
```

2.3 Other regularization methods

- Take inputs and distort them: rotate, flip, offset;
- Stopping before overfitting begins to occur.

SECTION 3
INITIALIZATION

3.1 Inputs

- When deciding on splitting a dataset, provided that there are more than 10000 entries, use the following proportions:
  - 98% train;
  - 1% valuation (testing during training);
  - 1% test (testing after training);
- Normalize average to 0:

$$\mu = \frac{1}{n}\sum x_i \qquad \Rightarrow \qquad \vec{x} = \vec{x} - \mu;$$

- Normalize variance to 1:

$$\sigma^2 = \frac{1}{n}\sum x_i^2 \qquad \Rightarrow \qquad \vec{x} = \vec{x}/\sigma^2.$$

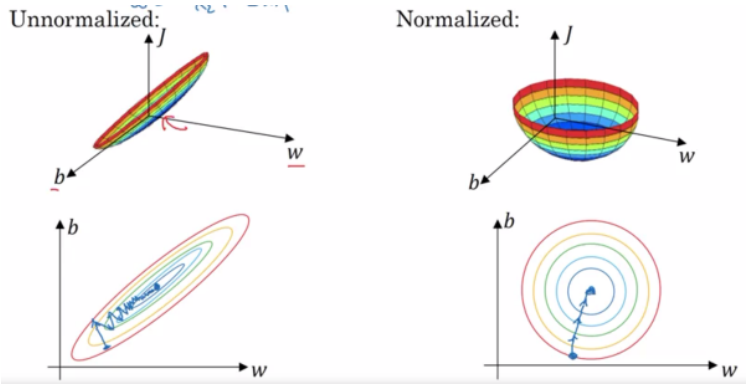


Figure 4: Normalization of inputs can lead to a preferable topology for gradient decent.

3.2 Weights

A NN must be initialized with weights from a normal distribution in order to improve learning efficiency. It has been shown that good performance is achieved when the distribution is normal and centered on 1:

$$N \sim (\mu = 0, \sigma = 1) \qquad \Rightarrow \qquad W_{\text{array}} = N(\text{dim-input}, \text{dim-output})$$

For the case when `RELU activation` is used, it is best to make the variance $2/m$ where m is the total number weights being used in the layer (this is known as `Xavier` initialization):

$$W_{\text{array}} = N(\text{dim-input}, \text{dim-output}) \times \sqrt{\frac{2}{\text{N}}}.$$

The effect of good weight initialization can be seen in Fig 5.

- Look at this documentation for more initializers of weights;
- $N \sim (0, 1)$ is the standard;

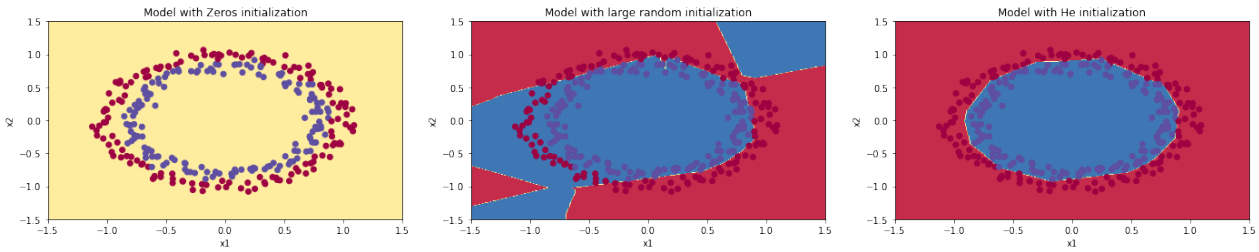


Figure 5: NN trained with different initial weights, left to right: **zero**, **random**, **Xavier**.

- $N \sim (0, \sqrt{2/N})$ is good when using `RELU activation`;
- Initializing weights when defining a layer in `keras`:

```
# 1 - initializing with normal
layer = ks.layers.Dense(60, kernel_initializer='normal')(PREVIOUS_LAYER)
# 2 - initializing with Xavier
layer=ks.layers.Dense(60,kernel_initializer=keras.initializers.glorot_normal(seed=
    None))(PREVIOUS_LAYER)
```

SECTION 4

Batch Normalization

`Batch Normalization` normalizes units in the hidden layers, first to $N(0,1)$ and then to $N(\eta_1, \eta_2^2)$:

$$\begin{cases} \mu = \frac{1}{n}\sum x_i \\ \sigma^2 = \frac{1}{N}\sum x_i^2 \end{cases} \xrightarrow{N\sim(0,1)} \vec{z} = \frac{\vec{x}-\mu}{\sigma^2+\epsilon} \xrightarrow{N\sim(\eta_1,\eta_2^2)} \vec{z}_n = \eta_1 + \eta_2 \times \vec{z}.$$

$\eta_{1,2}$ and ϵ (used to avoid division by 0) become hyperparameters of the network, which are also adjusted doing gradient descent i.e. $\frac{\partial J}{\partial \eta_i}$ and $\frac{\partial J}{\partial \epsilon}$ are evaluated and used in gradient descent: $\eta_i = \eta_i - \alpha * \frac{\partial J}{\partial \eta_i}$.

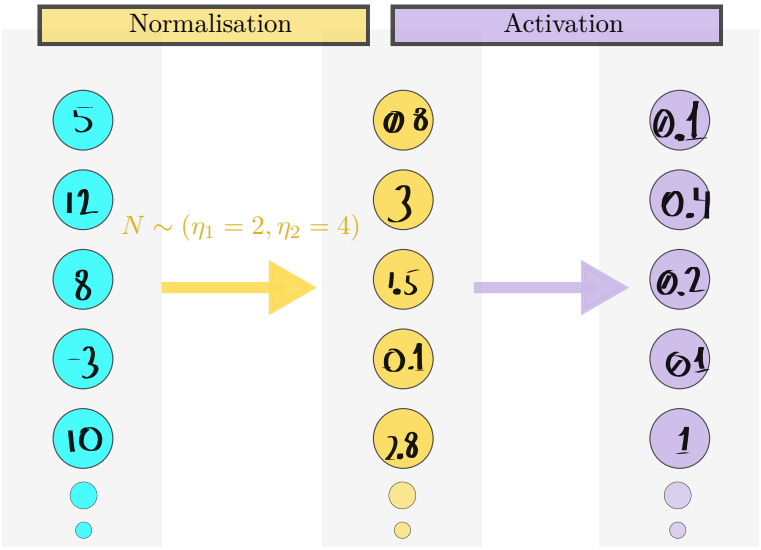


Figure 6: Normalization of the hidden layer should happen before activation.

- `Batch Normalization` makes the values of intermediate layers more stable;
- Weights that are adjusted/learned in earlier layers do not have strong repercussions in later ones, preventing exploding or vanishing values;
- **$\eta_{1,2}, \epsilon$ used in `Batch Normalization` become hyperparameters of the NN**;
- Inserting a `Batch Normalization` between `Dense` layer and it's `Activation Function`:

```
# 1 - insert a Dense Layer
PREVIOUS_LAYER = ks.layers.Dense(100)(PREVIOUS_LAYER)
# 2 - perform Batch Normalization
PREVIOUS_LAYER = ks.layers.BatchNormalization()(PREVIOUS_LAYER)
# 3 - now perform activation
PREVIOUS_LAYER = ks.layers.LeakyReLU(0.1)
```

SECTION 5

Minibatch Processing

When the NN is being trained, it will perform forward propagation of the full training set before evaluating the cost function and doing `gradient descent` (see Fig. **??** for a memory refresh). It is good practice to split the training set into batches, to make the network learn more reactively.

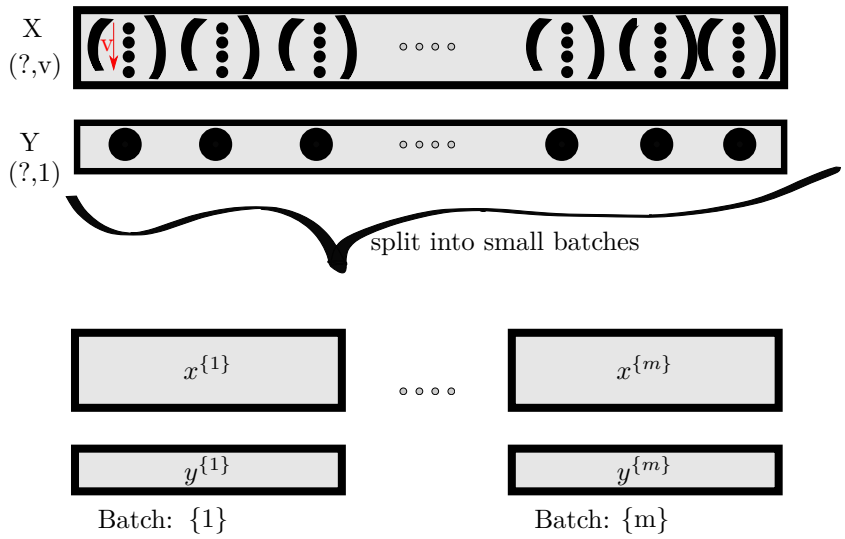


Figure 7: Weights will be updated using `gradient descent` after every minibatch.

The smaller the minibatches used, the more jagged the history of the cost function, and the more jagged the movement towards the local minimum of the cost function. The benefit is that you can track the progress of training more reactively, without having to wait for the NN to process all of the inputs before updating the weights. Very small batch sizes are also not ideal, as it would lead to sequential processing of the inputs (nn performs the full forward and backward propagation with a small amount of inputs), rather than the quicker parallel matrix multiplication.

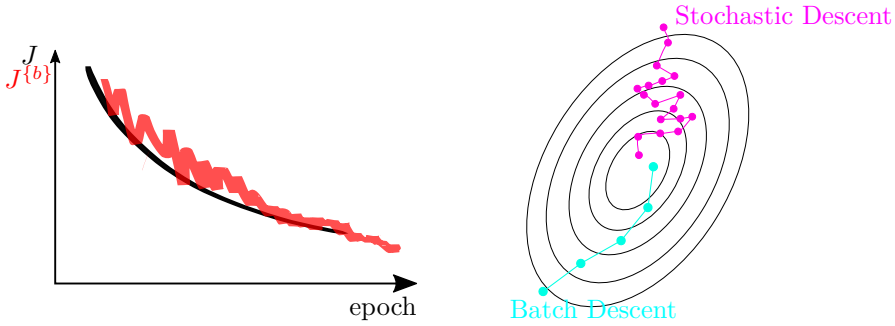


Figure 8: Minimization of the cost function happens with every batch. Smaller batches would update the value more frequently.

- When deciding on the batch size to use, consult the following table:

	Number of samples	Comment
`Batch descent`	$\leq 2000 samples$	Full sample set is iterated - Slow
`Stochastic descent`	1	Single sample are iterated - slow as you cannot parallelize processing
`Minibatch descent`	64, 128, 256, $\cdots$	

- **Control the minibatch size**;
- Training the model using a defined number of `minibatches`:

```
# 1 — set the size of the minibatches
minibatch_size = 1024
# 2 — train the model with the chosen minibatch size
train_history = model.fit(X, Y, epochs=1000, batch_size=minibatch_size)
```

SECTION 6

Optimizing Gradient Descent

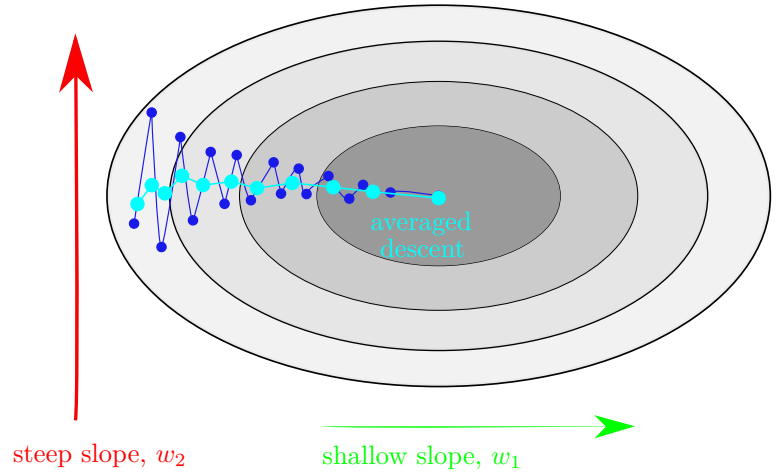
During each `epoch` the NN does:

1. **Forward propagation** to find the cost function, J;
2. **Backward propagation** to find the derivative of the `cost function` w.r.t. weights of the NN, $dw_i = \frac{\partial J}{\partial w_i}$;
3. **Gradient descent** to adjust the weights according the the aforementioned dependence, $w_i = w_i - \alpha \frac{\partial J}{\partial w_i}$.

It is in the interest of speed, to use a big `learning rate`, α, to adjust the weights as much as possible, "rolling" then down the hill to the local minimum:

$$w_i = w_i - \alpha \frac{\partial J}{\partial w_i}.$$

However a large `learning rate` means that on approach to the local minimum of the `cost function`, components with a big gradient undergo strong lateral oscillations, stalling optimization.



There are two tricks to fight this:

- Average out the oscillations by taking the average over the past few points, using exponentially weighted averages;
- Normalize by the mean square size of the oscillations.

6.1 Exponentially Weighted Averages

We will perform `gradient descent` on 1 particular weight w_i in the NN, using the shorthand notation $w \equiv w_i$. During backward propagation the NN takes the derivative of the `cost function` w.r.t. this weight

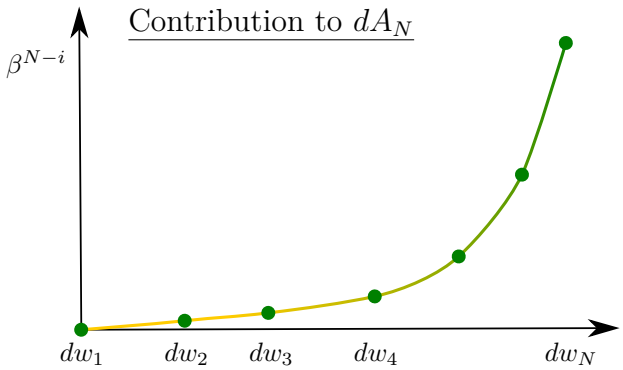
$$dw = \frac{\partial J}{\partial w};$$

Given history these derivatives, $dw_{1,2,3,\ldots}$, the running average, A, is computed with the iterative formula

$$dA_t = \beta dA_{t-1} + (1-\beta) dw_t.$$

dA_0	0	
dA_1	$\beta \mathrm{dA}_0$ + (1-β)dw_1	(1-β)dw_1
dA_2	$\beta dA_1 + (1-\beta) dw_2$	$\beta(1-\beta) dw_1 + (1-\beta) dw_2$
dA_3	$\beta dA_2 + (1-\beta) dw_3$	$\beta^2(1-\beta) dw_1 + \beta(1-\beta) dw_2 + (1-\beta) dw_3$
$\vdots$	$\vdots$	$\vdots$
dA_N		$(1-\beta) \sum_{i=t}^{N} \beta^{N-t} dw_t$

- Ultimately it is a weighted average over the recent history, with older gradients getting exponentially less significant;



- The bigger the momentum β, the more history past gradients are taken e.g.

$$\textit{for } \beta = 0.9 \textit{ the last 10 gradients are significant}$$
$$\textit{for } \beta = 0.99 \textit{ the last 100 gradients are significant.}$$

- As it is likely that the first couple of data points will be ≈ 0, a bias correction is used to boost the first couple of values, to "get out of 0":

$$A_t = \frac{(\beta A_{t-1} + (1-\beta)dw_t)}{(\mathbf{1}-\beta)^{\mathbf{t}}}.$$

- **Exponentially weighted averaging will smear the oscillations in the gradient descent. β (momentum) controls the extent of smearing.**

6.2 Improved gradient descent

In summary, the standard gradient descent now becomes:

1. **Forward propagation** to find the cost function, J;
2. **Backward propagation** to find the derivative of the `cost function` w.r.t. weights of the NN, $dw_i = \frac{\partial J}{\partial w_i}$;
3. Evaluation (for each dw_i) of the running average, A_t, of this derivative and it's mean square, σ_t:

$$dA_t = \frac{\beta_1 dA_{t-1} + (1-\beta_1)dw_i}{1-\beta_1^t} \qquad \sigma_t = \frac{\beta_2 \sigma_{t-1}^2 + (1-\beta_2)\left|dw_i\right|^2}{1-\beta_2^t}$$

4. **Gradient descent** to adjust the weights by the **averaged** and **normalized** history of gradients:

$$dw_i = dw_i - \alpha \frac{dA_t}{\sqrt{\sigma_t}}.$$

   - t is the step number;
   - β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
   - α is the learning rate
   - ε is a very small number to avoid dividing by zero

This `gradient descent` that has both averaging and normalization is known as "Adams Optimizer". Using this optimizer drastically improves training.

- Gradient descent for a given weight, w_i is optimized by:

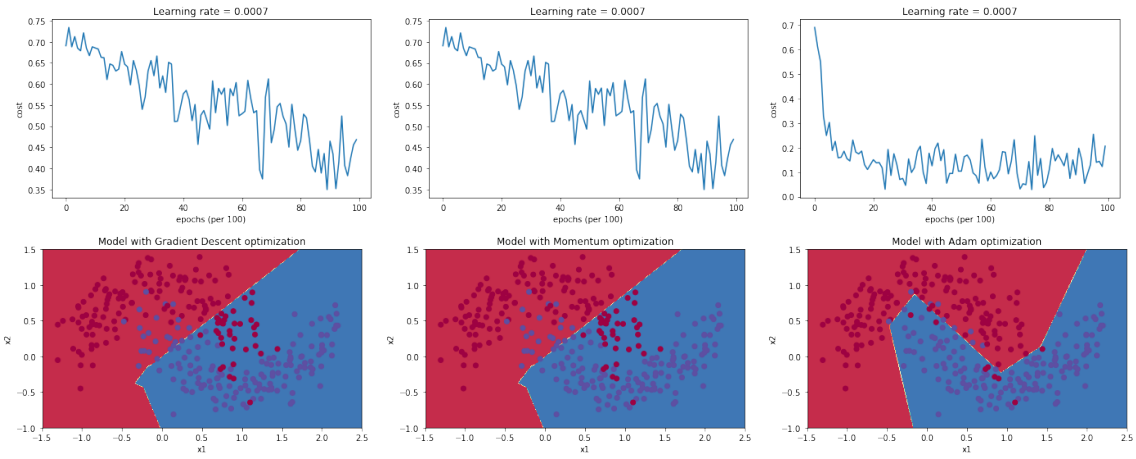


Figure 9: Left to right: **Standard Gradient Descent**, **Gradient Descent with exponential averaging**, **Adam's gradient descent**.

- Averaging over the history of derivative;
- Normalizing by the historical spread of the derivatives;

- Which makes the descent to the local minimum of the `cost function`, J, more direct;
- **Learning rate alpha** controls the rate of gradient descent;
- Momentum $\beta_1 \approx 0.9$ **controls the time lag of averaging the derivatives**;
- $\beta_2 \approx 0.99$ **controls the time lag of averaging the spread of the derivatives**;
- Other optimizers can also be chosen

```
# 1 - compile model with just the averaging
alpha=0.1
beta = 0.9
optimizer = SGD(lr=alpha, momentum=beta,)
model.compile(loss='binary_crossentropy', optimizer=optimizer)

# 2 - compiling model with deafault adam optimizer (averaging and normalization)
model.compile(loss=ks.losses.categorical_crossentropy, optimizer='adam')

# 3 - compile adam optimizer with tuned parameters
adam_tuned = keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999)
model.compile(loss=ks.losses.categorical_crossentropy, optimizer=adam_tuned)
```

SECTION 7

Learning Rate Decay

One brute force way of improving training is to manually set a decay rate for the learning rate α. The intention is to slow down once the local minimum is found. An example decay could be

$$\alpha = \frac{1}{1 + \text{decay rate} * \text{epoch number}}.$$

This is quite often not needed however.

```
adam_tuned = keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, decay =
    0.05)
model.compile(loss=ks.losses.categorical_crossentropy, optimizer=adam_tuned)
```

SECTION 8

Softmax Regression

Usually an activation function takes a value, and maps it non-linearly onto a value between 0 and 1.

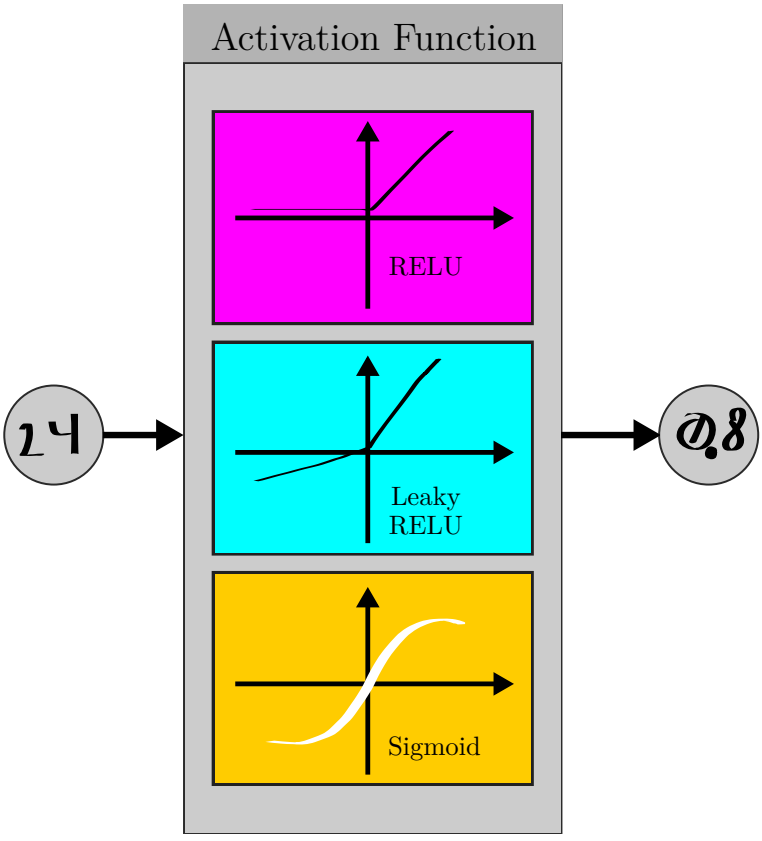


Figure 10: Activation functions map a neuron value 1-to-1.

However, suppose that a NN outputs multiple units, corresponding to different classes, which we want to convert to relative probabilities:

`Softmax regression` is used to scale the outputs to give probabilities by normalising by their magnitude relative to the others:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix} \Rightarrow \texttt{Soft Max} \Rightarrow \frac{1}{\sum_i \exp[y_i]} \begin{pmatrix} e^{y_1} \\ e^{y_2} \\ e^{y_3} \\ \vdots \end{pmatrix}.$$

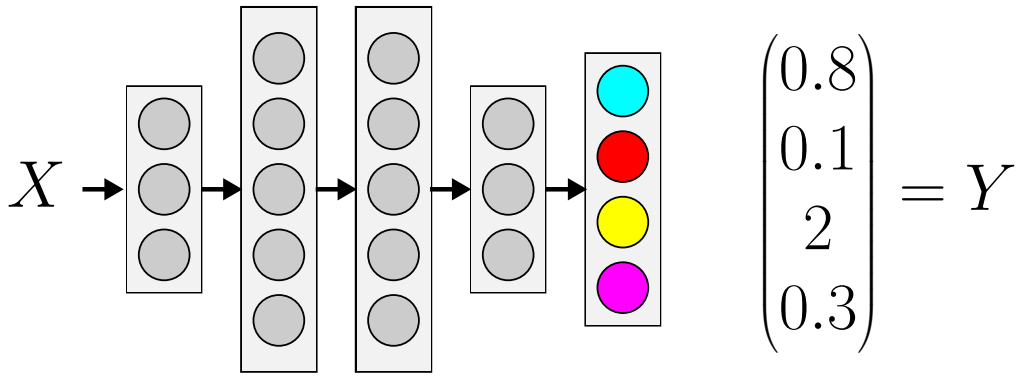


Figure 11: NN outputs units for 4 different classes, which we want to represent as probabilities

The cost function for this multi-class output reads

$$J = -\sum y_j \log \hat{y}_j \qquad y = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \end{pmatrix} \qquad \hat{y}_= \begin{pmatrix} 0.2 \\ 0.1 \\ 0.5 \\ 0.2 \\ \vdots \end{pmatrix}.$$

- Using `softmax regression` in `keras`:

```
SOFTMAX_LAYER= ks.layers.Dense(120, activation='softmax')(PREVIOUS_LAYER)
```

SECTION 9

HYPERPARAMETER REVIEW

So to conclude, we review the options that can be applied to some or all layers to improve the performance of a NN.

- All of the `hyper parameters` will be trained by the NN, by means of adjusting their value during back propagation e.g.
$$\eta_1 = \eta_1 - \alpha \frac{\partial J}{\partial \eta_1}.$$
- `Layers` can be added as stand-alone layers of the NN.
- The following methods can be used to improve performance:

Method	Description	Parameters
L1/L2 Regularization	Prevents overfitting by suppressing large weights	λ - strength of regularization
`Dropout`	Randomly shuts down neurons, to prevent the NN from overspecializing on a small subset	p_{dropout} - proportion of neurons to shut down each `epoch`
Distorting inputs	Flip, rotate, distort inputs to add noise to training	
Normalizing inputs	Normalize inputs to $N \sim (0, 1)$	
Initializing weights	Initialize weights to $N \sim (0, 1)$ or $N \sim (0, 2/m)$	Distribution to sample from
Minibatch processing	Split training set up, so that weights are updated more frequently	Minibatch size
Gradient Descent	Smooth the decent by taking averages over past gradients and normalizing by their variance	α - learning rate (speed of descent) β_1 - amount of averaging to perform β_2 - amount of averaging to perform on the variance
Learning rate decay	Specify the rate at which α should decrease	Rate of decay
`Batch Normalization`	Normalize units to $N \sim (\eta_1, \eta_2)$ in the hidden layers so that they are more stable. η_1 and η_2 become `hyperparameters` of the NN	
`Sofmax regression`	Turn outputs corresponding to multiple classes to relative probabilities	

- Parameters should be tuned in the following order of importance:

α	General feature
$\beta_1 \approx 0.9$, $\beta_2 \approx 0.99$, $\epsilon \approx 10^{-7}$	Gradient Descent
$\lambda \approx 0.01$	L1/L2 Regularization
$0.2 \leq p_{\text{dropout}} \leq 0.5$	`Dropout`
Size of hidden layers	General feature
Size of minibatch $\sim 2^n$	Minibatch processing
Learning rate decay	Learning rate decay
Number of hidden layers	General feature

- Parameters should be sampled from a non-linear distribution to probe sensitive regions.

  e.g. In ***gradient descent*** $\beta_1 = 0.9$ *will average over the past 10 samples, while* $\beta = 0.99$ *averages over the past 100. Thus the sensitive region is around 0.9, so sampling should be performed from a* log *distribution.*

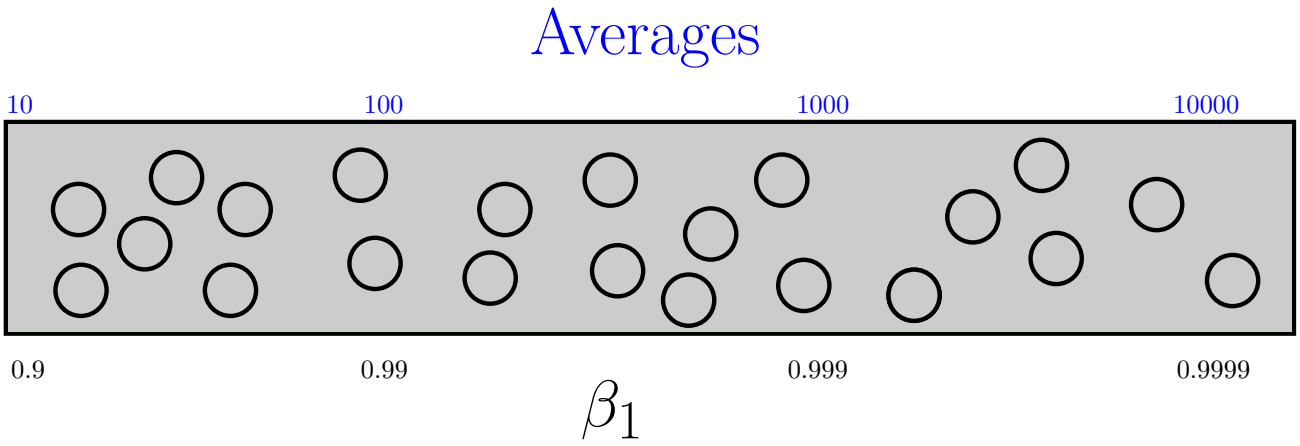


Figure 12: Sampling β_1 values to give advantage to the more sensitive region.