Deep Learning

Universidad Panamericana

David Esparza Alba

February 17, 2023

Contents

List of Figures

List of Programs

1

Introduction

Artificial Neural Networks (ANNs) come from simulating or replicating human neurons. They have been around us for many years, but in the last decade (maybe more), their applications have been boosted due to the high computational capacity and technological advances.

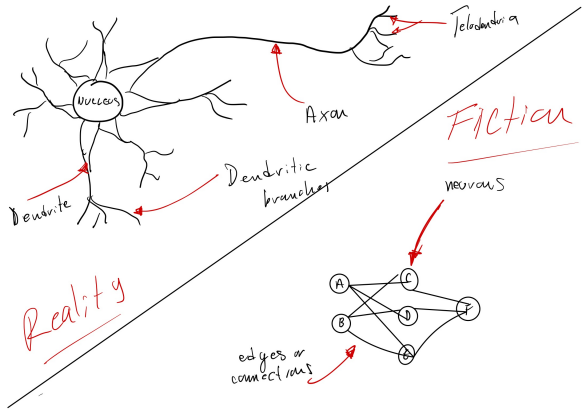


Figure 1.1: Artificial Neural Network

Neurons send electric pulses that "activate" other neurons. To simulate this, we use activation functions. The first activation function suggested were step functions, such as the **heavside** function and the **sign** function; both of them are step functions.

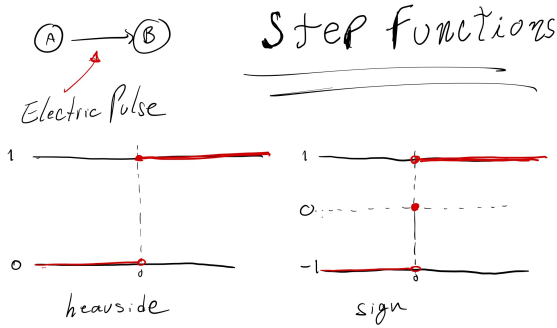


Figure 1.2: Step Functions

The goal consists of estimating the parameters representing the connections between neurons; these connections send information from one layer of neurons to another and are called weights or connection weights.

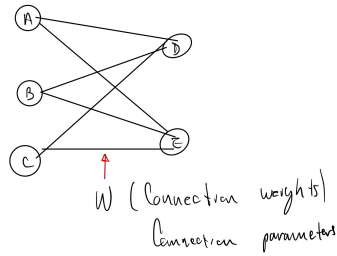


Figure 1.3: Step Functions

To estimate the weights parameters in Neural Network (NN), we use the Maximum-Likelihood estimation, which consists of finding the first derivatives of a **Loss Function** for each parameter we want to estimate. Calculating the derivatives was a drawback to using step functions since they are not differentiable.

In 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper introducing the **backpropagation** training algorithm. This algorithm uses gradient descent to

estimate optimal weight parameters, and to do this; they suggested a differentiable activation function. The **sigmoid** function.

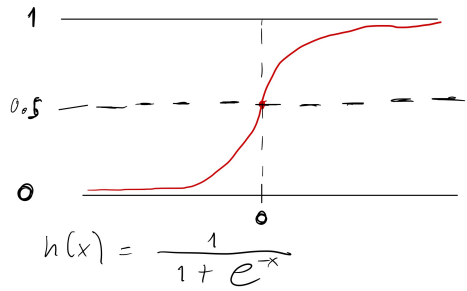


Figure 1.4: Step Functions

In a classification problem, the sigmoid function represents a probability of belonging to a particular class; if the value of $h(x) > 0.5$, it is classified as 1; otherwise, it is classified as 0. Following this concept, we can represent the classification problem as a Bernoulli experiment, so the probability distribution is given by

$$f(x, y) = h(x)^y (1 - h(x))^{1-y}$$

where $y \in \{0, 1\}$. The likelihood function is the product of independent and identical distributed samples.

$$\mathcal{L}(X, Y) = \prod_{i=1}^{n} h(x_i)^{y_i} (1 - h(x_i))^{1-y_i}$$

The log-likelihood is defined as follows.

$$\ell(X, Y) = \sum_{i=1}^{n} \left(y_i \log h(x_i) + (1 - y_i) \log(1 - h(x_i)) \right) \tag{1.1}$$

The equation 1.1 is the loss function for the Logistic Regression model for classification, and we will also use it to estimate the weight parameters of a Neural Network.

We usually use the negative log-likelihood to minimize the loss function. If we directly use our input in equation 1.1, we end up with a convex function, which causes gradient descent to converge to the global optimum.

In the case of NN, where there are multiple connections and layers, the loss function is not convex, so in theory, the gradient descent algorithm can converge to a local optimum, but in practice, NNs have shown good results.

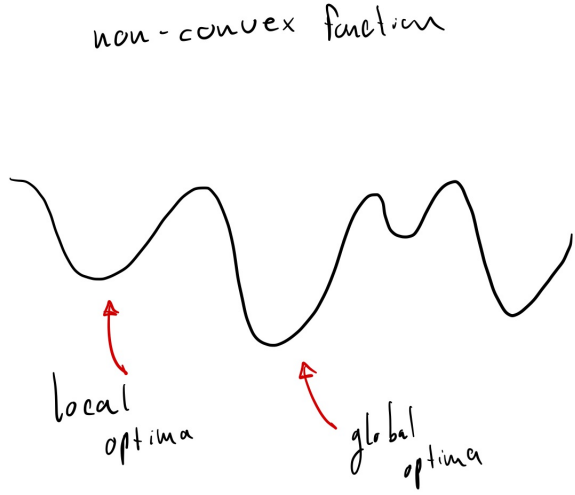


Figure 1.5: Step Functions

1.1 Setting Up Our Environment

1. Install miniconda in your computer.
2. Go to your working directory and open a terminal
3. Create a new environment.

```
>> conda create --name myenv
```

4. Activate your environment.

```
>> conda activate myenv
```

5. run the following command

```
>> conda install python=3.8 jupyter matplotlib numpy
 pandas scipy scikit-learn
```

6. Launch the jupyter server (port: 8888).

```
>> jupyter notebook
```

7. Go to your browser (http://localhost:8888/) and you will see a page similar to the following.

Figure 1.6: Jupyter Home Directory

8. Install TensorFlow.

```
>> conda install -c conda-forge tensorflow
```

1.2 The Gradient Descent Algorithm

Given a starting point x_0, we calculate the gradient, ∇, at x_0. The gradient is the vector with the first derivatives for each dimension or parameter. For the next iteration, we move in the direction of the gradient; then, we have the following formula.

$$x_{i+1} = x_i - \eta \nabla(x_i) \tag{1.2}$$

In this equation:

- $\nabla(x_i)$ is the gradient evaluated at x_i.

- η is the learning rate.

In figure 1.7, we notice that the solution moves in the gradient direction. The step size is called **learning rate**, and it is critical to define a proper value for it. A small learning rate will cause it to move slowly to the minimum; however, a large learning rate can cause it to pass the minimum and keep bouncing around it. In both cases, it may never reach the minimum. See figure 1.8.

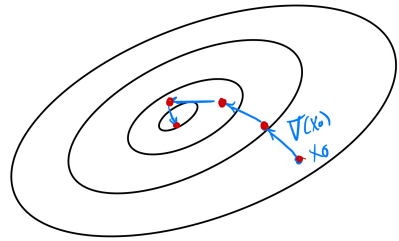


Figure 1.7: Iterations of Gradient Descent

Later we will mention some strategies that adapt the learning rate as the number of iterations increases.

There are other optimization methods to estimate the weight parameters; some use second-order derivatives, the Hessian matrix, but they are computationally expensive, so gradient descent is the most common algorithm for neural networks.

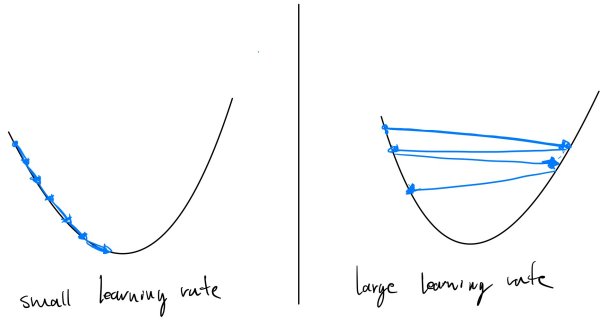


Figure 1.8: Learning Rate

1.3 The Backpropagation Algorithm

The backpropagation algorithm is the foundation for training neural networks, and it allows us to estimate the derivatives of the weight parameters to use gradient descent. To exemplify the functionality of the backpropagation algorithm, we will use the following 2-layer network.

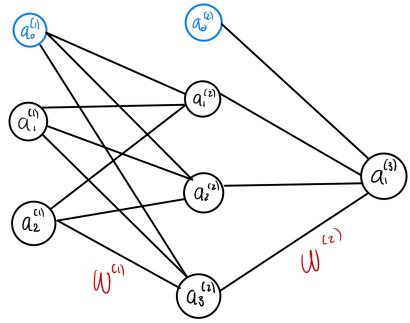


Figure 1.9: Neural Network with one input layer, one hidden layer, and one output layer

The input layer has two features or dimensions, such as an

(x, y) coordinate. The hidden layer contains three neurons, while the output layer consists of one. Each neuron of the k^{th}-layer is represented with the superscript (k). The neuron with the color blue represents the bias term of the linear equations, and the values of a for all the layers except the input layers come from the activation function, which for this example, is the sigmoid function, simulating a classification problem. We can represent the same network in 1.9 as follows.

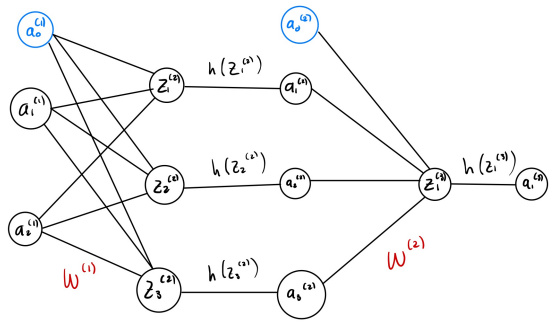


Figure 1.10: Neural Network with activation function

The goal is to estimate the values of $W^{(1)}$ and $W^{(2)}$. Something helpful is to keep in mind the dimensions of the weight matrices; if layer k contains n neurons, and layer $k+1$ contains m neurons, then the weight matrix representing the connections between both layers has a size of $m \times (n+1)$. Knowing the dimensions facilitates math calculations and coding.

The first step of the backpropagation algorithm is to execute a forward pass from the input layer to the output layer. The elements of the vector $z^{(2)}$ are defined below.

$$
\begin{aligned}
z_1^{(2)} &= W_{1,0}^{(1)} a_0^{(1)} + W_{1,1}^{(1)} a_1^{(1)} + W_{1,2}^{(1)} a_2^{(1)} \\
z_2^{(2)} &= W_{2,0}^{(1)} a_0^{(1)} + W_{2,1}^{(1)} a_1^{(1)} + W_{2,2}^{(1)} a_2^{(1)} \\
z_3^{(2)} &= W_{3,0}^{(1)} a_0^{(1)} + W_{3,1}^{(1)} a_1^{(1)} + W_{3,2}^{(1)} a_2^{(1)}
\end{aligned}
$$

We can vectorize these results and obtain

$$z^{(2)} = W^{(1)} a^{(1)}$$

Then we use the values of $z^{(2)}$ and the activation function, h, to obtain the values of the $a^{(2)}$.

$$a^{(2)} = h(z^{(2)})$$

Following the same process, we have:

$$z^{(3)} = W^{(2)} a^{(2)}$$

$W^{(2)}$ is a 1×3 matrix, and $a^{(2)}$ is a 3×1 vector. Then $z^{(3)}$ is a constant (1×1). We obtain the output from the network by passing $z^{(2)}$ through the activation function.

$$a^{(3)} = h(z^{(3)})$$

Here we have finished the forward pass through the network; then, we need to do a backward pass from the output layer to the input layer to obtain the gradients of the weight matrices, starting with $W^{(2)}$ and then with $W^{(1)}$.

We want to obtain the following:

$$\nabla(W^{(2)}_{j,i}) = \frac{\partial \ell}{\partial W^{(2)}_{j,i}} \tag{1.3}$$

Using the chain rule, we can express the equation 1.3 as follows.

$$\frac{\partial \ell}{\partial W^{(2)}_{j,i}} = \frac{\partial \ell}{\partial z^{(3)}_j} \frac{\partial z^{(3)}_j}{\partial W^{(2)}_{j,i}} \tag{1.4}$$

The term $\frac{\partial \ell}{\partial z^{(3)}_j}$ is called the **error term** and is represented as $\delta^{(3)}_j$. The value of $\frac{\partial z^{(3)}_j}{\partial W^{(2)}_{j,i}}$ is equal to $a^{(2)}_i$, so the equation of the derivative is the following.

$$\frac{\partial \ell}{\partial W^{(2)}_{j,i}} = \delta^{(3)}_j a^{(2)}_i \tag{1.5}$$

To obtain the value of $\delta^{(3)}_j$ we need to calculate $h'(z^{(3)}_j)$, which is the derivative of the sigmoid function on $z^{(3)}_j$.

$$
\begin{aligned}
h'(z_j^{(3)}) &= \frac{e^{-z_j^{(3)}}}{\left(1+e^{-z_j^{(3)}}\right)^2} \\
&= \frac{1+e^{-z_j^{(3)}}-1}{\left(1+e^{-z_j^{(3)}}\right)^2} \\
&= \frac{1}{1+e^{-z_j^{(3)}}} - \frac{1}{\left(1+e^{-z_j^{(3)}}\right)^2} \\
&= h(z_j^{(3)}) - \left(h(z_j^{(3)})\right)^2 \\
&= a_j^{(3)}(1-a_j^{(3)}) \qquad (1.6)
\end{aligned}
$$

The loss function evaluated on a specific input is the following.

$$\ell(z_j^{(3)}) = y \log h(z_j^{(3)}) + (1-y)\log(1-h(z_j^{(3)})) \qquad (1.7)$$

To obtain the error term, $\delta_j^{(3)}$, we need to calculate the derivative of this equation over $z_j^{(3)}$.

$$
\begin{aligned}
\frac{\partial \ell}{\partial z_j^{(3)}} = \delta_j^{(3)} &= y\frac{h'(z_j^{(3)})}{h(z_j^{(3)})} + (y-1)\frac{h'(z_j^{(3)})}{1-h(z_j^{(3)})} \\
&= \frac{-yh'(z_j^{(3)})h(z_j^{(3)}) + yh'(z_j^{(3)}) - h'(z_j^{(3)})h(z_j^{(3)}) + yh'(z_j^{(3)})h(z_j^{(3)})}{(1-h(z_j^{(3)}))h(z_j^{(3)})} \\
&= h'(z_j^{(3)})\frac{y-h(z_j^{(3)})}{(1-h(z_j^{(3)}))h(z_j^{(3)})} \qquad (1.8)
\end{aligned}
$$

Using equation 1.6 in 1.8 and replacing $h(z_j^{(3)})$ with $a_j^{(3)}$.

$$
\begin{aligned}
\delta_j^{(3)} &= a_j^{(3)}(1-a_j^{(3)})\frac{y-a_j^{(3)}}{(1-a_j^{(3)})a_j^{(3)}} \\
&= y - a_j^{(3)} \qquad (1.9)
\end{aligned}
$$

Then the derivative of the loss function over $W_{j,i}^{(2)}$ is given by

$$\begin{aligned}\frac{\partial \ell}{\partial W_{j,i}^{(2)}} &= \frac{\partial \ell}{\partial z_j^{(3)}} \frac{\partial z_j^{(3)}}{\partial W_{j,i}^{(2)}} \\ &= \delta_j^{(3)} a_i^{(2)} \\ &= (y - a_j^{(3)}) a_i^{(2)} \end{aligned} \tag{1.10}$$

To obtain the $\delta_j^{(2)}$ we need to express the loss function 1.7 in terms of $z_j^{(2)}$ and calculate its derivative. After doing some math, we obtain the following result.

$$\delta^{(2)} = W^{(2)T} \delta^{(3)} \cdot h'(z^{(2)}) \tag{1.11}$$

Summarizing, the backpropagation algorithm for a neural network with k layers consists of these steps:

1. As part of the forward pass through the network, obtain the values of z and a in each layer.

$$\begin{aligned} z^{(l)} &= W^{(l-1)} a^{(l-1)} \\ a^{(l)} &= h(z^{(l)}) \end{aligned}$$

2. Calculate the error term for the output layer (layer $k+1$).

$$\delta^{(k+1)} = y - a^{(k+1)}$$

3. For the rest of the layers, the error term is given by

$$\delta^{(l)} = W^{(l)T} \delta^{(l+1)} \cdot h'(z^{(l)})$$

With

$$h'(z^{(l)}) = a^{(l)} \cdot (1 - a^{(l)})$$

4. We proceed to compute the gradient of the weight parameters.

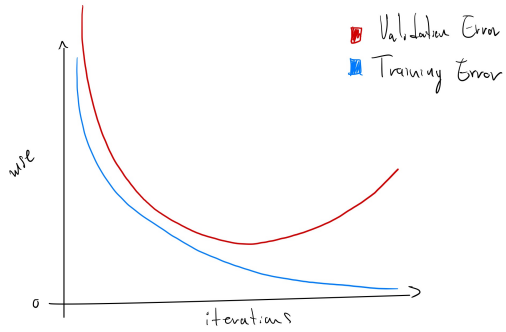
$$\nabla(W^{(l)}) = \delta^{(l+1)} a^{(l)T}$$

5. Finally, we can execute another iteration of the gradient descent algorithm using the gradients.

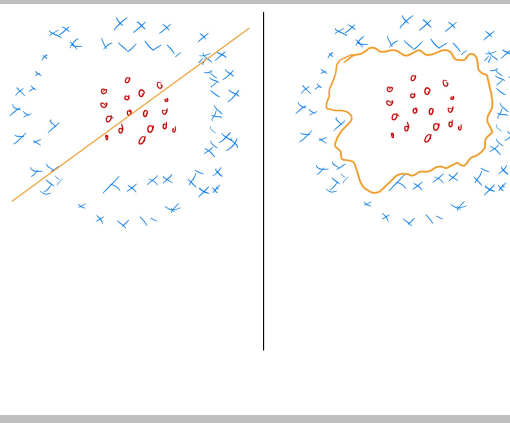
$$W^{(l)} = W^{(l)} - \eta \nabla (W^{(l)})$$

1.4 Exercises

1. Why can't we use the sign and step functions in a Neural Network?
2. Why is "sometimes" preferred to use *Gradient Descent* than other methods like Newton, or LBFGS.
3. What happens in Gradient Descent if the learning rate is too low or too high?
4. In a multi-layer neural network with multiple connections between layers and their neurons, the loss function is convex or non-convex?
5. What algorithm allows us to obtain the gradients of the connection weights while training a neural network?
6. In a classification problem, why do we use the *Cross-Entropy* loss function?
7. In the context of a classifier, define what is *accuracy*, *precision*, and *recall*.
8. The graph below represents the *MSE* of a regression model over multiple iterations. How would the model behave for new unseen data?



9. We image below displays the decision boundary of two classifiers for the same data; can you describe what is happening in both cases?



10. For a multi-class classification problem, what activation function would you use for the output layer and why?

11. Enumerate all the steps of the Backpropagation algorithm.

2

Neural Networks With Keras

2.1 Creating a Neural Network

For this example, we will use the Fashion MNIST dataset, which consists of a training set of 60000 28×28 images. The input layer consists of $28 \times 28 = 784$ units. We will create a network with two hidden layers with the *"relu"* activation function and one output layer with ten units, which is the number of categories of the dataset.

For the output layer, we will use the *"softmax"* activation function, which uses the multinomial method to support multiple classes and returns the result as a vector of probabilities.

Figure 2.1: Fashion MNIST Dataset

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28], name="input"),
    keras.layers.Dense(300, activation="relu", name="hidden1"),
    keras.layers.Dense(100, activation="relu", name="hidden2"),
    keras.layers.Dense(10, activation="softmax", name="output")
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================

 input (Flatten)              (None, 784)               0

 hidden1 (Dense)              (None, 300)               235500

 hidden2 (Dense)              (None, 100)               30100

 output (Dense)               (None, 10)                1010

=================================================================

Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

2.2 Functional API

You can create layers as functions, passing as a parameter the input, which is helpful for creating non-sequential networks.

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
flatten = keras.layers.Flatten(name="input")(input_)
hidden1 = keras.layers.Dense(300, activation="relu", name="
    hidden1")(flatten)
hidden2 = keras.layers.Dense(100, activation="relu", name="
    hidden2")(hidden1)
output_ = keras.layers.Dense(10, activation="softmax", name="
    output")(hidden2)

model = keras.Model(inputs=[input_], outputs=[output_])
model.summary()
```

```
Model: "model_12"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================

 input_18 (InputLayer)        [(None, 28, 28)]          0

 input (Flatten)              (None, 784)               0
```

```
 hidden1 (Dense)              (None, 300)               235500

 hidden2 (Dense)              (None, 100)               30100

 output (Dense)               (None, 10)                1010

=================================================================

Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

2.3 Callbacks

Callbacks execute certain tasks during training; it can happen at different times, for example, when an epoch finishes, when training finishes, etc.

The `EarlyStopping` function checks for the validation score after each epoch, and if there is no improvement after a certain number of epochs, then it stops the training. Early Stopping is helpful to avoid overfitting.

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
    restore_best_weights=True)
```

```
class MyCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print(f"\nval/train: {logs['val_loss'] / logs['loss']}")
```

```
history = model.fit(X_train,
                    y_train,
                    epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[early_stopping_cb, MyCallback()]
                   )
```

2.4 Tuning Hyperparameters

We can take advantage of *sklearn* cross-validation methods to fine-tune the parameters of a neural network. We need a wrapper of a Keras regressor, or classifier, and pass as a parameter a function that creates and compiles the neural network.

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=1e-3,
    input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation='relu
    '))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)

    return model
```

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(
    build_model)
```

3

Deep Learning Networks

bla bla bla

3.1 Vanishing and Exploding Gradients

Occasionally, gradients get smaller as the algorithm progresses, causing the lower layers' connection weights to remain unchanged, and the training never converges; this is called the **vanishing gradients** problem. When the opposite happens, and the gradients grow bigger and bigger, the algorithm can also diverge since the connection weights get incredibly large; this is the **exploding gradients** problem.

Using the sigmoid function, when the input is on the extremes, the function saturates, and the gradient is almost zero; this causes that during backpropagation, there is virtually nothing to propagate, and the lower layers remain unchanged.

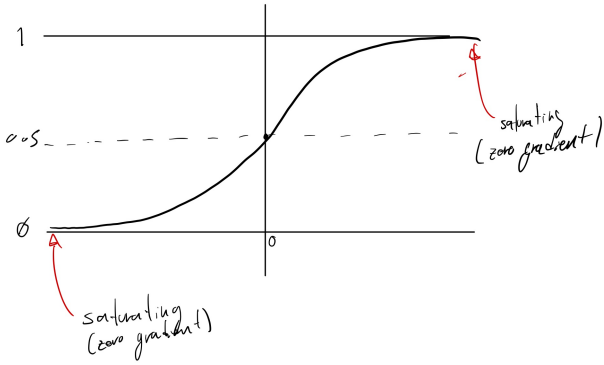


Figure 3.1: Saturating Sigmoid Function

Because of the vanishing and exploding gradients, Deep Neural Networks (DNN) were almost abandoned in the early 2000s. Still, in 2010, Xavier Glorot and Yoshua Bengio published a paper explaining that the activation function and the weight initialization could be causing problems with the gradients. They described that using the sigmoid function and a standard initialization causes the output variance of each layer to be much greater than the input variance, which is worse in the case of the sigmoid function, which has a mean of 0.5.

Glorot and Bengio proposed an initialization approach that works well in practice. The connection weights of each layer must be initialized randomly as described in equation 3.1, where $fan_{avg} = (fan_{in} + fan_{out})/2$, and fan_{in} and fan_{out} represent the number of inputs and neurons in the layer, respectively. This strategy is called the *Glorot initialization.*

$$\text{Normal distribution with mean 0 and variance } \sigma^2 = \frac{1}{fan_{avg}}$$
$$\text{Uniform distribution between [-r, +r], with } r = \sqrt{\frac{3}{fan_{avg}}} \tag{3.1}$$

Using Glorot initialization can speed up the training considerably, which is one of the reasons that led to the success of Deep

Learning. Other strategies have been proposed over the years, using different activation functions and initialization strategies; see table 3.1.

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

Table 3.1: Initialization for different activation functions

3.2 Activation Functions

Different activation functions have appeared over the years to deal with the saturating problem of the sigmoid function. One of these functions is the *ReLU* function, which doesn't saturates for non-negative values, and is always zero for negative values. Still, the *ReLU* has a problem, known as the *dying ReLUs*, which causes some neurons' output always to be 0. Figure 3.2 show the *ReLU* activation function and some of its variants.

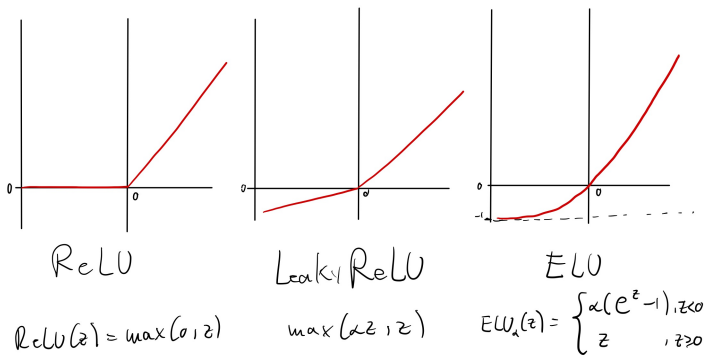


Figure 3.2: ReLU activation function and variants

To deal with the *dying ReLUs* problem, some variants have been proposed, like the *leaky ReLU*, which adds the hyperparameter α, ensuring that the *leaky ReLU* never dies. Another variant is the

ELU, which includes an exponential function for negative values that allows an average output closer to 0. It has a nonzero gradient for $z < 0$, avoiding the problem of dead neurons.

In 2017, the *Scaled ELU* (SELU) activation function was introduced, which is a scaled variant of the ELU activation function. Its authors showed that if we build a neural network containing a stack of dense layers with all hidden layers using the SELU activation function, the network will **self-normalize**. Still, there are some conditions for self-normalization:

- The input features must be standardized (mean 0 and standard deviation 1).
- The connection weights of every hidden layer must be initialized with *LeCun* normal initialization.
- The network must be sequential.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28], name="input"),
    keras.layers.Dense(300, activation="elu", kernel_initializer
    ="he_normal", name="hidden1"),
    keras.layers.Dense(100, activation="elu", kernel_initializer
    ="he_normal", name="hidden2"),
    keras.layers.Dense(10, activation="softmax", name="output")
])
model.summary()
```

3.3 Optimizers

Training a deep neural network can be quite slow, and because of that, new techniques have emerged over the years to boost the performance of gradient descent.

3.3.1 Momentum Optimization

The idea behind momentum optimization is to take larger steps as we continue moving to the minimum. The algorithm consists of adding the momentum vector, m, and the gradient to update the parameters. See figure 3.3.

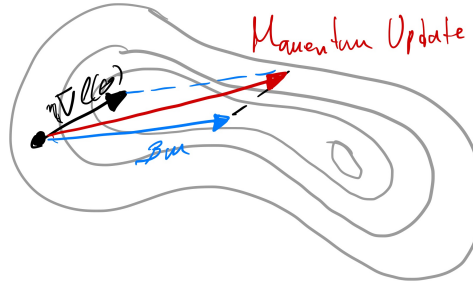


Figure 3.3: Momentum Optimization

The parameters, θ, are updated according to equation 3.2.

$$\begin{aligned} m &= \beta m - \eta \nabla(\theta) \\ \theta &= \theta + m \end{aligned} \tag{3.2}$$

```
optimizer = keras.optimizers.SGD(learning_rate=0.001, momentum
    =0.9)
```

3.3.2 Nesterov Optimization

The Nesterov method is similar to momentum optimization. Still, instead of using the gradient at the current location, it uses the gradient at the momentum's location to boost the convergence to the minimum. See figure 3.4

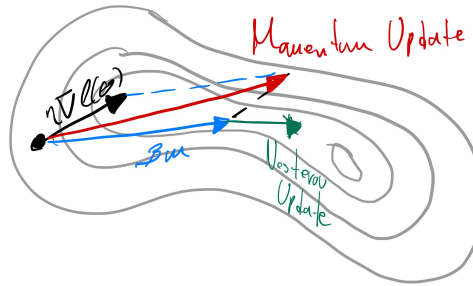


Figure 3.4: Nesterov Optimization

The parameters are updated with equation 3.3.

$$\begin{aligned} m &= \beta m - \eta \nabla(\theta + \beta m) \\ \theta &= \theta + m \end{aligned} \tag{3.3}$$

```
optimizer = keras.optimizers.SGD(learning_rate=0.001, momentum
    =0.9, nesterov=True)
```

3.3.3 RMSProp

Stores the gradients from the most recent iterations. The algorithm decays the learning rate so that it decays faster for steep dimensions. The parameters are updated as shown in equation 3.4.

$$\begin{aligned} s &= \beta s - (1 - \beta)\nabla(\theta) \cdot \nabla(\theta) \\ \theta &= \theta - \eta \nabla(\theta) \oslash \sqrt{s + \epsilon} \end{aligned} \tag{3.4}$$

The symbol $\oslash$ indicates the element-wise division, and ϵ is the smoothing term to avoid division by zero. The hyperparameter β represents the decay rate.

```
optimizer = keras.optimizers.RMSprop(learning_rate=0.001, rho
    =0.9)
```

3.3.4 Adam Optimization

The *Adaptive Moment Estimation* (Adam) algorithm combines the momentum optimization with the RMSProp optimization. It keeps track of an exponential decay average of the past gradients and an exponential decay average of past squared gradients. See equation 3.5.

$$\begin{aligned} m &= \beta_1 m - (1 - \beta_1)\nabla(\theta) \\ s &= \beta_2 s - (1 - \beta_2)\nabla(\theta) \cdot \nabla(\theta) \\ \hat{m} &= \frac{m}{1 - \beta_1^t} \\ \hat{s} &= \frac{s}{1 - \beta_2^t} \\ \theta &= \theta - \eta \hat{m} \oslash \sqrt{\hat{s} + \epsilon} \end{aligned} \tag{3.5}$$

The term t represents the iteration number, starting at 1.

```
optimizer = keras.optimizers.Adam(learning_rate=0.001, beta_1
    =0.9, beta_2=0.999)
```

3.4 Learning Rate Scheduling

3.4.1 Power Scheduling

The learning rate drops at each iteration and is defined as follows.

$$\eta(t) = \frac{\eta_0}{(1 + t/s)^c} \tag{3.6}$$

The initial learning rate η_0, s, and c are hyperparameters.

```
optimizer = keras.optimizers.SGD(learning_rate=0.01, decay=1e-4)
```

The function's parameter, `decay`, represents the inverse of s.

3.4.2 Exponential Scheduling

The learning rate is defined as a function of the iteration number.

$$\eta(t) = \eta_0 0.1^{t/s} \tag{3.7}$$

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)

lr_scheduler = keras.callbacks.LearningRateScheduler(
    exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[lr_scheduler])
```

3.4.3 Performance Scheduling

Measure the validation error every N step, and reduce the learning rate by a factor of λ when the error stops dropping. The callback below multiplies the learning rate by 0.5 if the validation loss doesn't improve for five consecutive epochs.

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5,
    patience=5)

optimizer = keras.optimizers.SGD(learning_rate=0.02, momentum
    =0.9)

history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[lr_scheduler])
```

3.4.4 1cycle Scheduling

Initially increases the learning rate η_0 linearly up to η_1 during the first half of the training. Then it decreases the learning rate to η_0 during the second half of training. The learning rate drops by several orders of magnitude in the last epochs.

3.5 Batch Normalization

The Vanishing/Exploding gradients problem can be reduced with proper initialization; it doesn't guarantee that the same problem won't happen later during training. In 2015 Christian Szegedy proposed a technique known as **Bath Normalization**; this technique consists of adding a normalization operation before or after the activation function of each hidden layer. This operation zero-centers and normalizes each input, followed by shifting and scaling steps. **In some cases, if we add a BN layer as the first layer, we don't need to standardize the input.**

`BatchNormalization` has become so popular in deep neural networks that it is occasionally omitted in diagrams.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Sometimes, using the BN layer before the activation function works better; to do this, we must set the parameter `use_bias` to `False`.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
```

```
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),
    keras.layers.Dense(100, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

3.6 Regularization

3.6.1 ℓ_1 and ℓ_2 Regularization

The ℓ_2 regularization is helpful to constrain the connection weights, while the ℓ_1 regularization is useful when we want a sparse model. The ℓ_2 norm represents the Euclidean distance of the connection weights, and the ℓ_1 norm represents the absolute value of the connection weights.

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

3.6.2 Dropout

At every training step, every neuron (except output neurons) has a probability p of being temporarily "dropped out" and will be ignored during this training step. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work?

We need to multiply each input connection weight by $(1-p)$ after training. We can divide each neuron's output by p during training.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer
    ="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer
    ="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

3.6.3 Monte Carlo Dropout

Dropout takes place during prediction as well. Averaging over multiple predictions with dropout turned on gives a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout turned off.

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)

class MCAlphaDropout(keras.layers.AlphaDropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

3.7 Summary

Hyperparameter	Default Value
Kernel Initializer	He initialization
Activation Function	ELU
Normalization	Batch normalization if deep
Regularization	Early Stopping (+ ℓ_2 if needed)
Optimizer	Momentum (or RMSProp or Nadam)
Learning rate scheduling	1cycle

Table 3.2: Default DNN configuration

Note: Nadam optimization consists of Adam optimization plus the Nesterov trick. The following table the default configuration for a self-normalizing DNN.

Hyperparameter	Default Value
Kernel Initializer	LeCun initialization
Activation Function	SELU
Normalization	None
Regularization	Alpha dropout if needed
Optimizer	Momentum (or RMSProp or Nadam)
Learning rate scheduling	1cycle

Table 3.3: Default DNN configuration

Note: For a self-normalizing network, don't forget to normalize the input features.

Bibliography