Deep Learning: Assignment #1

Alejandro Gómez de Miguel - CIT Student ID: [redacted]

08/03/2020

Introduction

This report covers the implementation and evaluation of Deep Learning models using two widely known frameworks for neural network algorithms: TensorFlow and Keras. Additionally, the final section of this document provides a comprehensive explanation of Adam, an adaptative learning rate optimization technique, where we go through the theoretical and practical aspects of it. Please note, that along with this report we've provided Python notebooks where you can find the code for the algorithms. These are written in Python 3 and all our experiments were run in Google Colaboratory using GPU hardware accelerator to speed up the executions.

1. TensorFlow and the Low-Level API

The first part of the project consists of implementing several neural network architectures using TensorFlow's low-level functions and evaluating their performance using the FashionMNIST dataset. This is an image classification problem, where the algorithms will be trained to learn the underlying patterns of different clothing items. More specifically, there are 10 class labels corresponding to 10 unique items. This dataset contains 60,000 and 10,000 grayscale images of 28x28 resolution for training and testing, respectively. However, during the experimentation phase, we will hold back 10% of the training set for validation. Additionally, note that we use Adam as optimizer and random initialization of weights based on a normal distribution with zero mean and standard deviation of 0.05. As part of the preprocessing endeavor, each of the images has been converted into a flattened array of 784 normalized pixels that take values between 0 and 1.

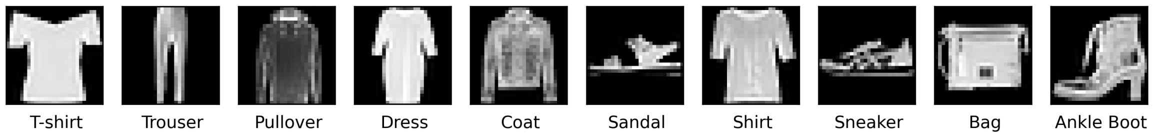


Figure 1: Randomly selected images representing each of the classes in the FashionMNIST dataset.

1.1 Softmax Classifier

A general approach when we aim to solve a classification problem is to build a baseline model that we can compare with more complex ones. A Softmax Classifier is a basic architecture consisting of an input and output layer. In other words, each of the output neurons that represent a class in the output layer is fully connected to every pixel of an incoming image in the input layer. As we're using 28x28 images and 10 classes, there are 7,840 connections established between the input and output layers - formally called weights. Additionally, there's one bias term per output neuron, resulting in a total sum of 7,850 parameters in the model.

There are four fundamental operations in neural networks: forward pass, loss calculation, accuracy, and backpropagation. Thanks to modern Deep Learning frameworks such as TensorFlow, the latest can be easily executed using tools that calculate the partial derivative of our loss function w.r.t. the parameters of the model, automatically updating these using the resulting gradients. In this report, we will focus our attention in the first three operations mentioned earlier. To be more specific, we discuss below in detail our vectorized implementation of the algorithm using the source code that we wrote to train it concerning the forward pass, the loss calculation, and the level of accuracy of the model.

```python
def forward_pass(features, weights, bias):

  # Matrix multiplication between the weights and feature values + bias
  pre_activated = tf.matmul(weights, features) + bias
  # Element-wise exponentiation
  activated = tf.math.exp(pre_activated)
  # Compute the total sum of activated values for a given instance
  sum_logits = tf.reduce_sum(activated, axis=0)
  # Transform activated values to conditional probabilities (normalization)
  probabilities = tf.math.divide(activated, sum_logits)
  # Clip 'probabilites' tensor to max/min values to avoid underflow
  probabilities = tf.clip_by_value(probabilities, 1e-10, 1.0)

  return probabilities
```

Code block 1: Forward pass function. Softmax Classifier.

The forward pass constitutes the entire journey of the feature matrix from the input layer to the output layer. In order to perform a vectorized implementation, all the feature data in the input layer is stored in a transposed 2-dimensional tensor of shape `[# features, # instances]`, where each row the gray intensity values of a specific pixel across all images, and each column shows the intensity value of every pixel in a single image i.e. each image is now represented as a column instead of a row. Similarly, as every neuron in the output layer is fully connected to the input layer, we build a 2-dimensional tensor of shape `[# classes, # features]`. In this tensor, each row constitutes the connections between an output neuron for a given class in the Softmax layer and the outputs of every neuron in the first layer. Each column is the number of connections between a single neuron in the first layer and all the neurons in the output layer. This is essentially the data structure that stores the weights of the model. Additionally, we also create a tensor with the biases of the model with shape `[# classes, 1]` - one bias per output neuron. The forward pass function in code block 1 takes these three inputs: feature data, weights, and biases. It performs the linear calculations of the model's hypothesis using matrix multiplication and adds the bias. This results in tensor p (`pre_activated`) with shape:

$$(weights : [c, x]) \cdot (features : [x, i]) + (biases : [c, 1]) = (p : [c, i])$$

where c is the number of neurons or classes in the output layer, x is the number of features or pixels in each image, and i is the total number of observations. Each row contains the pre-activated values of a single neuron representing a class for all the input data. These values are passed through the exponential activation function in the Softmax layer, which adjusts and converts all these pre-activated values to positive float numbers by performing element-wise exponentiation (`activated`). Finally, the neuron that outputs the highest activated value for a given image will be the model's predicted class. However, to make these results practically interpretable, they are normalized so that the final output is the conditional probability of an image being classified as class c given feature data x (`probabilities`).

$$p(y = c|x) = \frac{e^{(W_y \cdot x + B_y)}}{\sum_{c=1}^{C} e^{(W_c \cdot x + B_c)}}$$

In our code, we divide the tensor of activated values by a rank 1 vector (`sum_logits`) with shape `[# instances]` derived from the column-wise summation of activated values per image. Lastly, to avoid potential issues with arbitrarily small probabilities, these are capped to not exceed a minimum and maximum value.

The next step is to calculate the Cross Entropy Total Loss so we can provide feedback to the algorithm during training to adjust the weights of the model. The Cross Entropy Loss calculation is shown in the equation below, where p is the true probability distribution, and q is the probability distribution of the Softmax Function.

$$L(p, q) = -\sum_{i=1}^{N} p(x_i) \log(q(x_i))$$

Intuitively, as the class labels are one-hot encoded in our dataset, $p(x_i)$ is one for the correct class and zero for the rest of classes. Hence, we can simplify this calculation further and express it as the negative log of the probability predicted for the true class. Regardless, we have implemented this calculation as closest to the underlying theory as possible for clarity.

```python
def cross_entropy(probabilities, labels):

  # - log(probability predicted for the correct class)
  loss = tf.reduce_sum(tf.math.multiply(tf.math.log(probabilities), (-labels)), axis=0)
  # Total loss is calculated as the average instance loss
  total_loss = tf.reduce_mean(loss, axis=0)

  return total_loss
```

Code block 2: Cross Entropy function. Softmax Classifier.

In code block 2, the cross entropy function takes two inputs: a tensor of predicted probabilities with shape `[# classes, # instances]`, and a matrix of one-hot encoded values representing the true distribution of the data with the same shape. Thus, we can perform element-wise multiplication between the log probabilities and the negative values of the true class labels. As a result, this will return a `[# classes, # instances]` matrix where most of the values have been canceled by zeros, and only the negative log probability of the correct class is present in each column for each image as a positive float number that represents the loss of the model's prediction. Additionally, we convert this into a rank 1 vector named `loss` by performing column-wise summation over the entire tensor. Lastly, the Cross Entropy Total Cost is simply calculated as the average instance loss that we store in `total_loss`.

During training, the algorithm is internally running an optimization problem that gradually minimizes the cost function. At each iteration, it performs the forward pass, calculates the overall loss, and, additionally, checks the current level of accuracy of the model. Code block 3 shows our implementation of the latest.

```python
def calculate_accuracy(probabilities, labels):

  # Get the index of the actual class from a one-hot encoded vector
  actual_class = tf.argmax(labels, axis=0)
  # Get the index of the predicted class with highest probability
  predicted_class = tf.argmax(probabilities, axis=0)
  # As the indices must be equal to the classes, we compare both tensors
  bool_acc = tf.math.equal(predicted_class, actual_class)
  # Correctly classified images divided by total images
  accuracy = tf.math.divide(tf.math.count_nonzero(bool_acc), actual_class.shape[0])

  return accuracy
```

Code block 3: Level of accuracy function. Softmax Classifier.

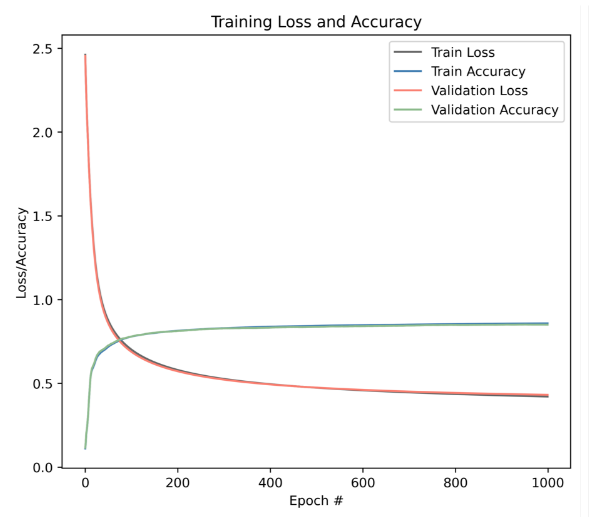
The inputs of the function are the predicted probabilities and true class labels of the images. In order to work with discrete data, we assume that the predicted class of the model for an incoming instance is the class associated with the highest probability. This allows us to extract from each tensor and for each observation the index position at which the function values are maximized, and store them as rank 1 tensors in `actual_class` and `predicted_class` with shape `[# instances]`. Due to the fact that the index positions must match with the class labels of the dataset, we are able to perform element-wise equality comparison between these two tensors, resulting in a boolean vector with the same shape indicating the matches found (`bool_acc`). Finally, we take advantage of how logical boolean operators work to count non zero values in this vector i.e. `True` values, and divide it by the total number of instances on which the model was run. This outputs the level of accuracy of the model given a set of probabilities and true classes.

Our experiments using the Softmax Classifier has shown that despite the model's simplicity, it's powerful enough to rapidly reach a reasonable level of accuracy without overfitting to the data.

Iteration	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
# 100	0.715	0.771	0.702	0.770
# 300	0.529	0.829	0.527	0.824
# 500	0.477	0.844	0.479	0.835
# 700	0.448	0.852	0.454	0.845
# 1000	0.422	0.86	0.434	0.852

Table 1: Training results of the Softmax Classifier.

Table 1 shows the learning state at different training stages. It's noticeable the exponential decay of the average instance loss at the first iterations, where the optimization algorithm moves quickly in the right direction. Gradually, the gradients and learning rates become smaller as the optimizer gets closer to the global minimum of the cost function. The chart on the left-hand side of Figure 2 provides a clear representation of the training process. The algorithm adjusts the parameters of the model while adapting properly to unseen data, achieving a similar level of accuracy at each iteration on both training and validation sets.



Actual \ Predicted	T-shirt	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle Boot
T-shirt	808	4	12	49	8	0	105	0	14	0
Trouser	5	952	4	26	6	0	5	0	2	0
Pullover	22	5	730	9	140	1	82	0	11	0
Dress	24	10	16	866	34	1	44	0	5	0
Coat	0	3	103	36	768	0	81	0	9	0
Sandal	0	1	0	0	0	893	0	60	2	44
Shirt	139	3	123	38	111	1	559	0	26	0
Sneaker	0	0	0	0	0	28	0	933	0	39
Bag	3	1	5	9	2	6	21	5	948	0
Ankle Boot	0	0	0	0	0	10	2	42	0	946

Figure 2: Softmax Classifier's loss and accuracy during training (Left) and confusion matrix on the test set (Right).

The accuracy on the test set is 0.84, which ultimately indicates that the model generalizes well to new observations However, we noticed that it's finding hard to learn how a shirt looks like. The confusion matrix in Figure 2 highlights the misclassification errors of the model on each class. More specifically, when the true class label is 'Shirt', the model predicts 'T-shirt', 'Pullover', and 'Coat' quite often. Figure 3 below shows randomly selected images included in the test set that the algorithm incorrectly classified.

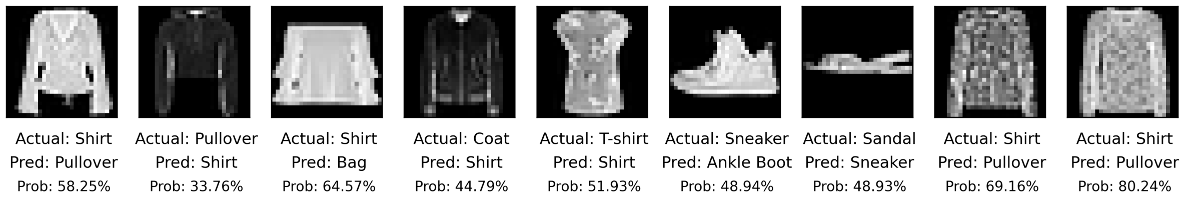


Figure 3: Randomly selected misclassified images on the FashionMNIST's test set.

For reference, each of the images also shows the probability computed in the Softmax layer for the predicted class. Some of the examples above constitute trivial errors, which makes us believe that a more powerful model should be able to deal with them. Regardless, the last two examples could be representative of the shirt issue, where the model is confident these are pullovers but they are actually shirts. However, we must say that potentially even a pair of human eyes wouldn't find easy to distinguish between these two.

1.2 Neural Networks

In this part, the Softmax Classifier is expanded adding new layers with Relu based neurons in between the existing input and output layers. Deeper architectures will drastically increase the number of parameters and complexity of the model, being able to achieve a higher level of accuracy on the FashionMNIST dataset. We developed a neural network with one hidden layer of 300 neurons that has 238,510 trainable parameters, and an extended version of this one with two hidden layers of 300 and 100 neurons, respectively, and 266,610 tunable parameters.

```python
def forward_pass(features, weights, bias):

  # ** First layer (ReLu) **
  pre_activated_h1 = tf.matmul(weights[0], features) + bias[0]
  # ReLu activation function: element-wise maximum of (y1', 0)
  activated_h1 = tf.math.maximum(pre_activated_h1, 0)

  # ** Second layer (Softmax) **
  pre_activated_out = tf.matmul(weights[1], activated_h1) + bias[1]
  # Element-wise exponentiation
  activated_out = tf.math.exp(pre_activated_out)
  # Compute the total sum of activated values for a given instance
  sum_logits = tf.reduce_sum(activated_out, axis=0)
  # Transform activated values to conditional probabilities (normalization)
  probabilities = tf.math.divide(activated_out, sum_logits)
  # Clip 'probabilities' tensor to max/min values to avoid underflow
  probabilities = tf.clip_by_value(probabilities, 1e-10, 1.0)

  return probabilities
```

Code block 4: Forward pass function. Multilayer Neural Network (1HL; 300 ReLu-based neurons).

This upgrade necessarily requires updating the forward pass function so that the image pixels now navigate through a deeper network and experience further transformations before predicting the class label of an image. Code block 4 shows the modified feed forward function of the neural network architecture with one hidden layer and 300 neurons.

Essentially, matrix multiplication is now executed not only in the Softmax layer but also in the additional hidden layer. The first layer takes the image pixels from the input layer and performs the linear hypothesis using the weights and biases of the first layer (`pre_activated_h1`). Then, as the neuron units that receive this information are ReLu-activated, if the value resulting from the linear hypothesis of the model is negative, the activated value is set to zero, otherwise, it passes through the activation function as it is with no transformations (`activated_h1`). This last operation is carried out in an element-wise fashion given a 2-dimensional tensor with shape `[# L1 neurons, # instances]`, which is the one that feeds the output layer. At this stage, the algorithm will execute exactly the same pipeline of operations that we described earlier with regards to the Softmax layer; from matrix multiplication using the weights and biases of the output layer resulting in a tensor of pre-activated values (`pre_activated_out`) with shape `[# L2 neurons, # instances]`, to element-wise exponentiation (`activated`) and normalization (`probabilities`) of these last ones.

Incorporating additional layers into the network is relatively straight forward and follows the same coding structure that we've seen earlier. Code block 5 contains the forward pass function of the multilayer neural network with two fully-connected layers of 300 and 100 neurons using ReLu activation. The main difference concerning the previous architecture, is just an additional calculation of the linear model's hypothesis along with the activation function between the first layer with 300 neurons and the Softmax layer with 10 output neurons. Consequently, the feature data is modeled three times sequentially before being reduced to probabilities.

```python
def forward_pass(features, weights, bias):

  # ** First layer (ReLu) **
  pre_activated_h1 = tf.matmul(weights[0], features) + bias[0]
  # ReLu activation function: element-wise maximum of (y1', 0)
  activated_h1 = tf.math.maximum(pre_activated_h1, 0)

  # ** Second layer (ReLu) **
  pre_activated_h2 = tf.matmul(weights[1], activated_h1) + bias[1]
  # ReLu activation function: element-wise maximum of (y2', 0)
  activated_h2 = tf.math.maximum(pre_activated_h2, 0)

  # ** Third layer (Softmax) **
  pre_activated_out = tf.matmul(weights[2], activated_h2) + bias[2]
  # # Element-wise exponentiation
  activated_out = tf.math.exp(pre_activated_out)
  # Compute the total sum of activated values for a given instance
  sum_logits = tf.reduce_sum(activated_out, axis=0)
  # Transform activated values to conditional probabilities (normalizatio
  probabilities = tf.math.divide(activated_out, sum_logits)
  # Clip 'probabilites' tensor to max/min values to avoid underflow
  probabilities = tf.clip_by_value(probabilities, 1e-10, 1.0)

  return probabilities
```

Code block 5: Forward pass function. Multilayer Neural Network (2HL; 300 and 100 ReLu-based neurons).

The increased complexity of these models yielded to higher accuracy rates and lower average instance losses, however, in this case, there are clear patterns showing that the networks are overfitting to the data. For instance, the validation loss at iteration 500 is lower than the one at iteration 1000, although it continually decreases in the training set. This gap between the performance of the model in the two sets of data highlights the presence of overfitting. Ultimately, the algorithm is fine-tuning the parameters of the model in such a way that it's starting to memorize this data instead of learning the underlying patterns.

Iteration	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
Single HL (300n)				
# 100	0.400	0.864	0.410	0.856
# 300	0.270	0.907	0.328	0.878
# 500	0.211	0.928	0.312	0.888
# 700	0.171	0.943	0.313	0.890
# 1000	0.129	0.960	0.327	0.891
Two HL, (300n, 100n)				
# 100	0.399	0.861	0.409	0.853
# 300	0.269	0.904	0.331	0.883
# 500	0.199	0.931	0.321	0.886
# 700	0.163	0.942	0.341	0.890
# 1000	0.112	0.963	0.370	0.891

Table 2: Training results of the Multilayer Neural Networks with 1 and 2 hidden layers.

Figure 4 shows the learning curves of the algorithms and the gap between the models' performance on the training and validation sets we just mentioned. In contrast to the Softmax Classifier, at a very early stage of the training process, the validation loss not only *plateaus* but also starts increasing gradually. This is especially noticeable for the more complex model with two hidden layers. Although they outperformed our baseline model where each one achieved a level of accuracy of 0.884 in the test set, they do not generalize well to unseen data, which is a basic requirement for scalable learning algorithms.

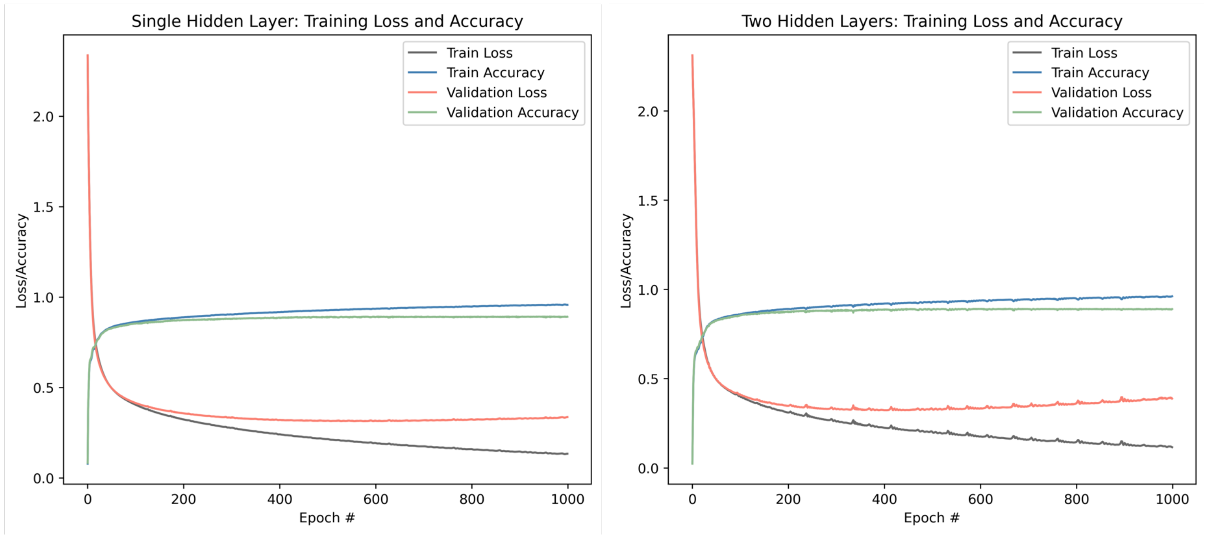


Figure 4: Loss and accuracy on training and validation data of neural networks with one (Left) and two (Right) hidden layers.

Regardless, the confusion matrices obtained after running these models on the test set (Figure 5), show that they have acquired skills to recognize shirts better reducing the number of misclassified images under this label. More specifically, there has been a noticeable improvement at differentiating between shirts, pullovers, and coats. However, they keep presenting a similar error rate for shirts that were predicted as t-shirts.

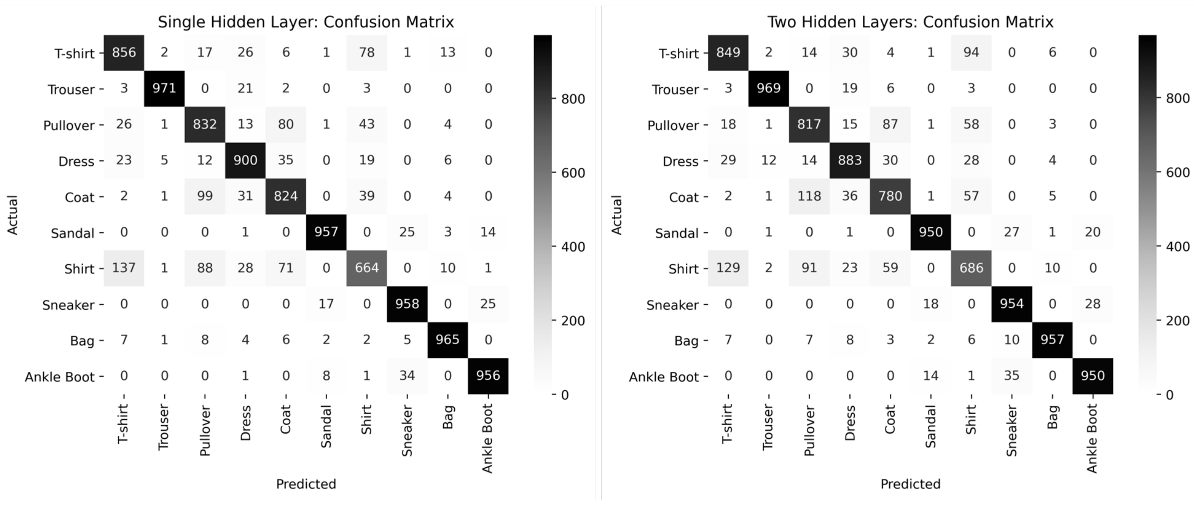


Figure 5: Classification results on the test set of neural networks with one (Left) and two (Right) hidden layers.

Finally, we conclude that there's no benefit from adding more layers of neurons into these networks. Even know the second architecture with two hidden layers with 300 and 100 neurons is theoretically a more powerful model, it didn't perform better than the model with a single hidden layer. Indeed, it overfits the training data much more easily, which contributes negatively to the classification problem.

1.3 L1 & L2 Regularization

Neural networks are very powerful models that can approximate any mathematical function by tuning its parameters. Overfitting is the ultimate proof of this capability, where the weights of the network are perfectly adjusted to fit the training data. This represents a problem in the study of Machine Learning and Deep Learning because these models poorly generalize to new observations. Regularization is a technique aimed at preventing the algorithms from adjusting the existing parameters in such a way that the resulting model is only able to recognize patterns in the training data. This leads to simpler models and representations that can also be applied to evaluate new observations. Regularization is practically incorporated in a model as a constrain in the loss function. In this part, we'll focus on discussing and implementing L1 and L2 regularization in our neural network with two hidden layers and 300 and 100 neurons each.

L1 Regularization: It aims to minimize the absolute value of all the weights of the model by adding a new term in the Cross Entropy Cost function. Generally, at each iteration, the optimization algorithm updates the parameters of the model so that the overall loss is less than the previous one and the values of the weights are the closest to zero as possible. Regularization is controlled using δ, where larger values of it penalize the model's weights greatly. It is expected that the algorithm will cancel non-significant features to reduce the overall loss i.e. setting them to zero, reducing the complexity of the model.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(y_i, \hat{y}_i) + \delta \sum_{i=1}^{m} |\theta_i|$$

L2 Regularization: This technique is incorporated in the Cross Entropy Cost function as well. However, instead of looking at the total sum of absolute values, it squares the parameters of the model. This strategy's main purpose is to penalize the algorithm even if it is keeping very few large values available in the parameter space. Due to the nature of exponentiation, this method aims to set weight values around zero too. As in L1 regularization, δ manages the term's contribution to the overall loss of the model.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(y_i, \hat{y}_i) + \delta \sum_{i=1}^{m} \theta_i^2$$

In order to incorporate this technique in the multilayer neural network code, we have updated the cross entropy function as follows:

```python
def cross_entropy(probabilities, labels, weights, regularization, delta):

  # - log(probability predicted for the correct class)
  loss = tf.reduce_sum(tf.math.multiply(tf.math.log(probabilities), (-labels)), axis=0)

  # Set regularization loss to 0
  reg_loss = 0.0

  if regularization == 'L1':
  # For every layer in the network with weights associated:
    for i in range(len(weights)):
    # (1) Compute square of w element-wise, (2) sum the entire vector,
    # (3) and add and assign to reg_loss
      reg_loss += tf.reduce_sum(tf.math.abs(weights[i]))
  elif regularization == 'L2':
  # For every layer in the network with weights associated:
    for i in range(len(weights)):
    # (1) Compute the absolute value of w element-wise, (2) sum the entire vector,
    # (3) and add and assign to reg_loss
      reg_loss += tf.reduce_sum(tf.math.square(weights[i]))

  # Total loss is calculated as the mean of the individual losses + regularization term
  total_loss = tf.reduce_mean(loss) + (delta * reg_loss)

  return total_loss
```

Code block 6: L1 and L2 regularization. Multilayer Neural Network (2HL; 300 and 100 ReLu-based neurons).

In code block 6, `weights` is a Python list containing as many TensorFlow's weight variables as layers in the model. In our deepest network, there are two hidden layers and one output layer, so the length of this list is equal to 3. The new function takes two additional arguments as inputs, the regularization technique to be applied as a string (L1 or L2), and the value of delta. It firstly calculates the individual cost entropy loss for each instance. Then, it initializes a global variable that will store the regularization loss during subsequent computations. Both regularization techniques are implemented using a `for` loop that iterates over the list containing the weights of each layer. When L1 is passed as an argument, the function adds all the absolute values of the weights and use the *add and assing* operator to store it in `reg_loss`. If L2 is selected, it adds all the squared weights computed element-wise and incrementally adds and stores the results in `reg_loss`. Lastly, the total loss is calculated as the sum of the average individual loss and the adjusted regularization loss where δ comes into play as a multiplier.

Throughout our experiments, we noticed that tuning δ effectively is critical towards a good performance of the model. The following metrics and charts correspond to δ values of 0.0001 and 0.002 for L1 and L2, respectively. However, we can't guarantee that these are optimal. Table 3 shows the learning states of the network at different stages during training, where we can appreciate the positive contribution of regularization to the problem of overfitting.

Iteration	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
L1 Regularization ($\delta = 0.0001$)				
# 100	0.941	0.846	0.945	0.838
# 300	0.583	0.878	0.605	0.869
# 500	0.487	0.890	0.525	0.875
# 700	0.433	0.901	0.485	0.880
# 1000	0.387	0.910	0.457	0.886
L2 Regularization ($\delta = 0.002$)				
# 100	0.790	0.849	0.794	0.844
# 300	0.530	0.879	0.548	0.868
# 500	0.497	0.886	0.522	0.872
# 700	0.478	0.893	0.509	0.879
# 1000	0.467	0.899	0.503	0.881

Table 3: Training results of the Multilayer Neural Network with 2 hidden layers and regularization.

Specifically for this network architecture, we previously highlighted that the validation loss was gradually increasing as the algorithm was run for more iterations. In this case, when regularization is applied, both training and validation losses decrease over time while the optimizer is fine-tuning the parameters of the model, which is the ideal learning behavior we would expect. This is represented graphically in Figure 6. Although there's a small gap between the training and validation loss when using L1 and L2, it has been greatly reduced from Figure 5 and the loss isn't increasing overtime anymore - it seemingly reached a region close to the global minimum of the cost function.

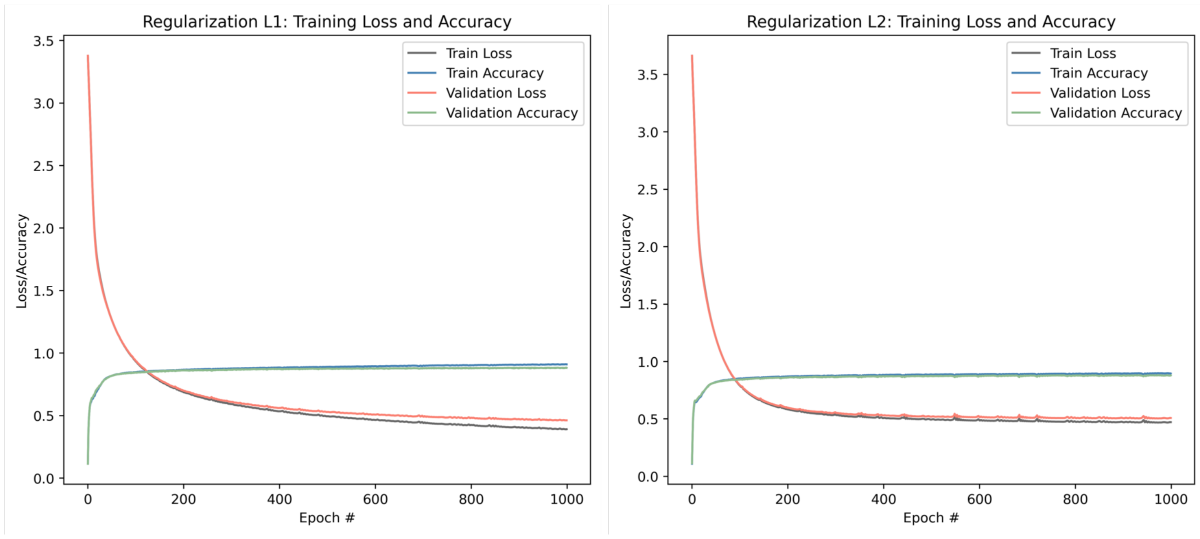


Figure 6: Loss and accuracy on training and validation data of the multilayer neural network using L1 (Left) and L2 (Right) regularization.

The level of accuracy of the network on the test has sightly decreased from 0.884 to 0.876 and 0.874 for L1 and L2, respectively. This is, however, expected since we're incorporating an extra cost in our loss function and reducing the search space. Regardless, these results provide evidence that a simpler and constrained model is able to achieve a performance almost identical to a complex and fully performance-based one. Additionally, we have noticed that the regularized models have been able to obtain enough knowledge to differentiate between shirts, pullovers, and coats as good as the multilayer neural networks from the previous section. Nevertheless, interestingly, regularization has shifted the errors between shirts and t-shirts. In other words, the non-regularized models showed a high error rate predicting t-shirt when the image was actually a shirt, while these models have reduced these cases but the number of t-shirts predicted as shirts significantly increased. Compare figure Figure 7 with Figure 5. Potentially, larger values of some specific weights were letting the network give more importance to t-shirts than shirts.

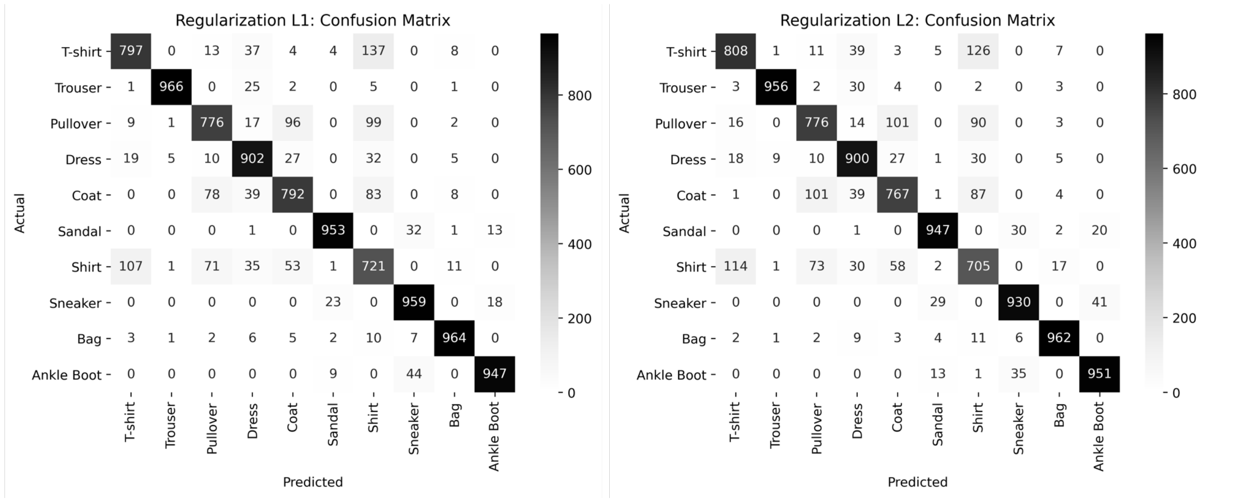


Figure 7: Classification results on the test set of the multilayer neural network using L1 (Left) and L2 (Right) regularization.

Lastly, we conclude this section with Figure 8. It shows randomly selected misclassification errors of the L2-regularized model on the test set, which is the one that, despite obtaining a slightly lower accuracy, we believe has proven to be a more robust model towards the goal of dealing with overfitting in neural networks.

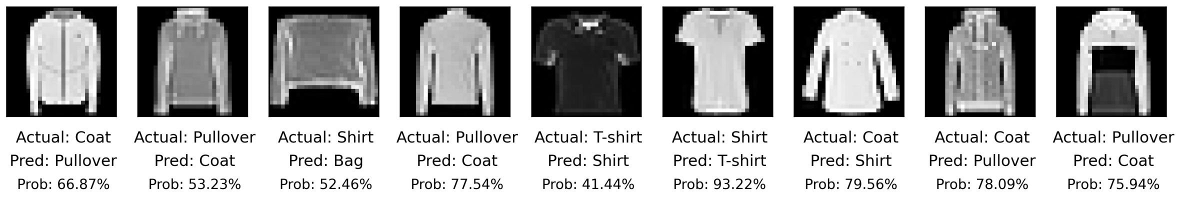


Figure 8: Randomly selected misclassified images on the FashionMNIST's test set.

2. Keras - High-Level API

Keras is presented as a high-level Deep Learning framework for prototyping and fast experimentation. It makes it more intuitive to run models and it doesn't require the user to have a detailed understanding of how neural networks work. In this part of the project, we will mainly focus on practical aspects of image classification problems and Deep Learning models. In particular, we will be training different neural network architectures on a character dataset called notMNIST.

This is a multiclass classification problem that consists of 10 labels, each one of these corresponding to symbols that represent English letters from A to J. The dataset contains 200,000 and 17,000 grayscale images of 28x28 resolution for training and test, respectively. Additionally, 10% of the training data is held back for validation. Each image has been flattened to a vector with 784 normalized pixels, which are now considered feature values of the dataset. Note that the weights of the models in this section are initialized using Keras default uniform initializer, the optimizer is set to Adam with default parameters, and batch size is 256 images per forward pass.

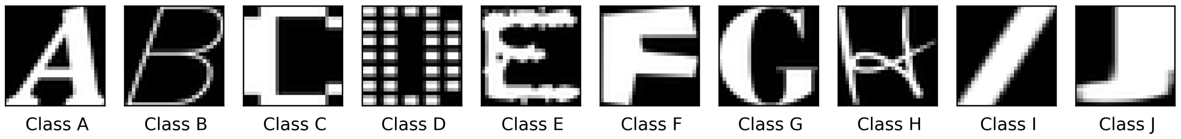


Figure 9: Randomly selected images representing each of the classes in the notMNIST dataset.

2.1 Building Neural Networks

The main goal of this part is to evaluate the performance of neural network architectures and report the observed improvements from adding more layers and neurons. As we previously did, a Softmax Classifier sets the baseline performance so that we can further compare it with more complex models. Table 4 summarizes the models' configurations we trained. It's worth noting the increasing complexity of the networks as more layers and neurons are added.

Model	Layers	Neurons	Number of Parameters
Softmax Classifier	(x1 Softmax)	(10)	7,850
Neural Network	(x1 ReLu, x1 Softmax)	(200, 10)	159,010
Neural Network	(x2 ReLu, x1 Softmax)	(400, 200, 10)	396,210
Neural Network	(x3 ReLu, x1 Softmax)	(600, 400, 200, 10)	793,430
Neural Network	(x4 ReLu, x1 Softmax)	(800, 600, 400, 200, 10)	1,431,030

Table 4: Technical description of a selection of neural network architectures.

We expect that most of the configurations above will rapidly overfit to the training data due to be working with a relatively simple dataset. Our experiments consisted in running each algorithm for 20 epochs, storing at each iteration the loss and accuracy on the training and validation sets. Furthermore, the resulting models make predictions on the test set, where we evaluate the level of accuracy and average individual loss.

The results obtained indicate that the Softmax Classifier is actually the only model that isn't trying to memorize the training data. In fact, even a neural network with a single hidden layer is showing this behavior. However, there's a significant improvement in the level of accuracy by just adding one layer. We also noticed that there's no evidence that a more complex model would be able to perform better on this problem. As a matter of fact, the network architectures with more than one hidden layer achieved approximately the same loss and accuracy on the test set.

Model	Loss	Accuracy
Softmax Classifier (x1 Softmax)		
Training	0.618	0.836
Validation	0.639	0.829
Test	0.550	0.854
NN (x1 Relu, x1 Softmax)		
Training	0.185	0.945
Validation	0.445	0.891
Test	0.345	0.912
NN (x2 Relu, x1 Softmax)		
Training	0.068	0.978
Validation	0.589	0.904
Test	0.445	0.923
NN (x3 Relu, x1 Softmax)		
Training	0.067	0.978
Validation	0.617	0.908
Test	0.474	0.925
NN (x4 Relu, x1 Softmax)		
Training	0.074	0.976
Validation	0.561	0.908
Test	0.446	0.925

Table 5: Models' performance on notMNIST dataset.

Figure 10 clearly illustrates overfitting and the performance similarity between certain models. At this stage, our investigation should focus on methodologies to overcome this training issue. Ultimately, a Softmax Classifier isn't appropriate for this classification problem due to its simplicity, but deep architectures return a level of accuracy that is biased due to limited generalization. In the next section, we cover a regularization technique aimed at building simpler models during training by limiting the networks' ability to fine-tune excessively the connections that help them memorize the data.

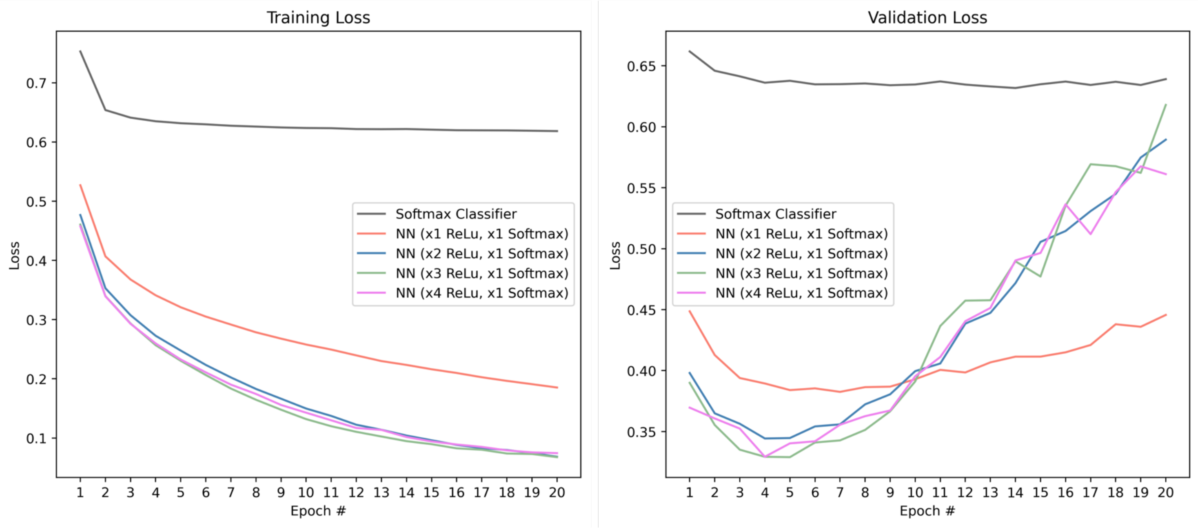


Figure 10: Training loss (Left) and validation loss (Right) for a selection of models.

2.2 Dropout Regularization

Deep neural networks establish millions of connections between neurons in different layers creating non-linear relationships that help the algorithms identify complex patterns precisely. During training, some of the neurons become very important for the network as the flow of calculations in a forward pass return minimum loss values. The network receives feedback regularly in the form of gradients thanks to backpropagation and modifies the weights accordingly to make these dependency relationships even stronger. This is the prowess of neural networks but also the reason why overfitting occurs.

Dropout is a regularization technique that addresses this problem by randomly dropping neurons from every layer in the forward pass during training with a certain probability. In other words, right after passing a value through the activation function of a neuron, the resulting output can become zero with p probability, which is equivalent to removing the neuron from the layer and its connections. In essence, this allows us to train an algorithm using a variety of randomly generated neural network architectures and prevent overfitting.

We have applied dropout to the two deepest networks evaluated in the previous section that showed a high degree of overfitting. Tuning effectively the probability of dropout p is of great importance in order to reach a reasonable balance between learning and overfitting. During our experiments, we observed that setting different dropout probabilities based on the number of neurons in each layer wasn't more beneficial than using a single and unique dropout rate across all the layers of the model. However, this might not be always the case for all networks' architectures and classification problems. Figure 11 shows the learning curves of the two neural networks selected in this section when dropout is applied.

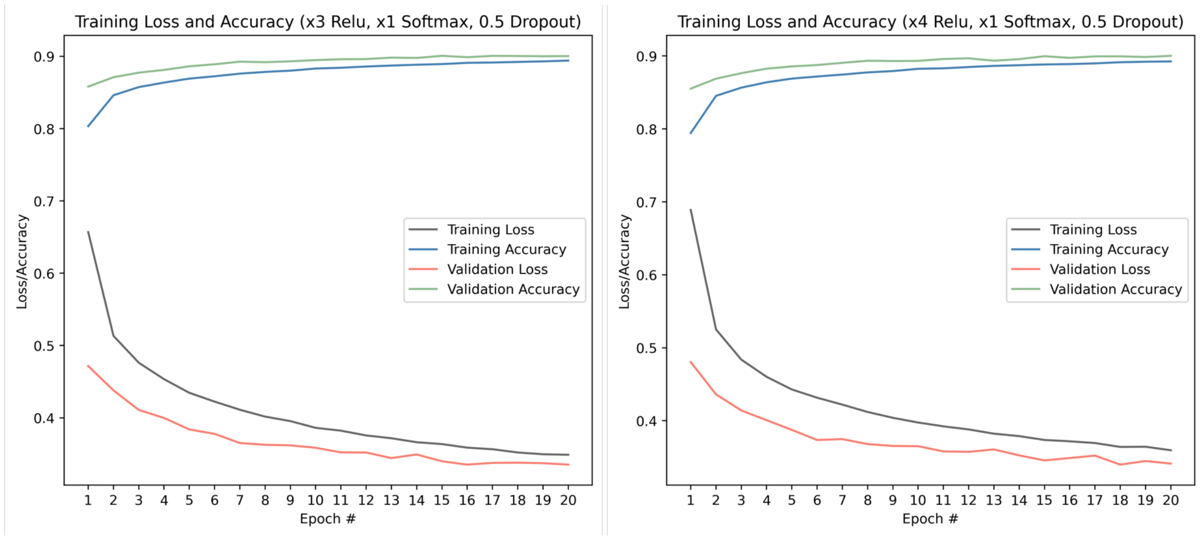


Figure 11: Loss and accuracy of deep neural network architectures using Dropout regularization.

It's worth looking back at Figure 10 to compare the huge improvement obtained after introducing regularization in the algorithms. The validation loss now follows the path of the training loss, which indicates that the networks generalize well to unseen data during training. The gap between these two metrics is progressively being closed over the iterations as the optimizer gets closer to the global minimum of the cost function. In our experiments, we found that values greater than 0.5 slowed down the optimization and they made the model so simple that the algorithm wasn't able to learn as much as it could. On the other hand, values lower than 0.3 weren't enough to overcome overfitting.

The level of accuracy on the test set after 20 epochs is 0.919 and 0.921 for the smaller and bigger model, respectively. Although obtaining the lowest error rate possible is out of the scope of this analysis, we observed that the algorithms slightly improved after more iterations resulting in test accuracy measurements close to 0.93. Figure 12 illustrates the outstanding performance of the models on the test set showing a strong diagonal in the confusion matrices.

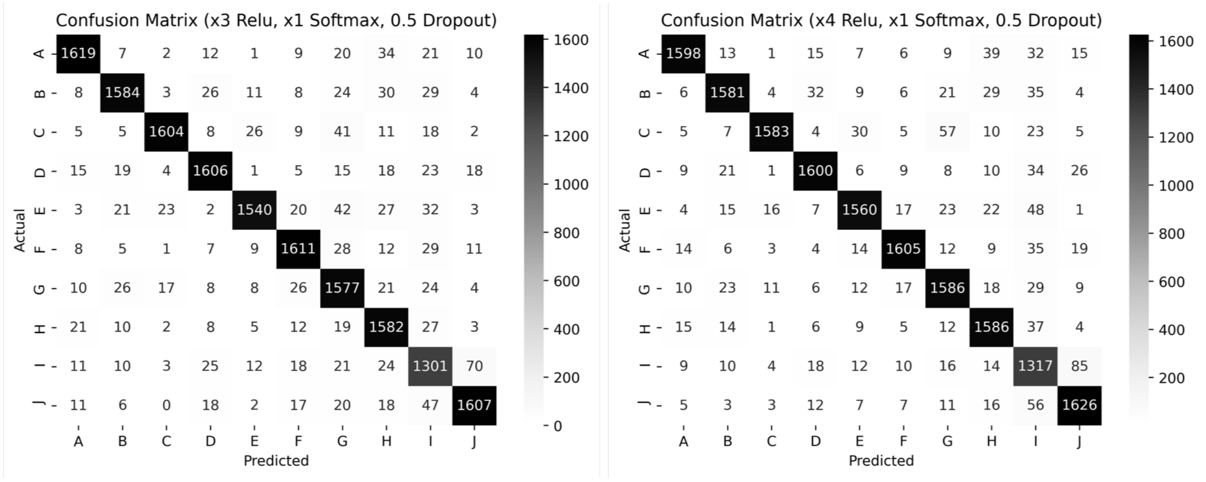


Figure 12: Classification results on the test of deep neural networks using Dropout regularization.

Lastly, we have investigated some of the errors of the best performing neural network (x4 Relu, x1 Softmax, Dropout (0.5)). It is interesting to see that some of these shouldn't even be considered errors as the instances are not even representing actual letters, or they are abstract representations that humans wouldn't recognize easily. However, there is an image showing a letter 'E' that the model predicted as 'B' with low probability. This is, indeed, because this type of neural networks learn patterns based on pixel intensity, which occasionally can lead to trivial errors on examples such as this one, where the pixel intensity has been inverted and now large values (black) indicate the lettering and desing.

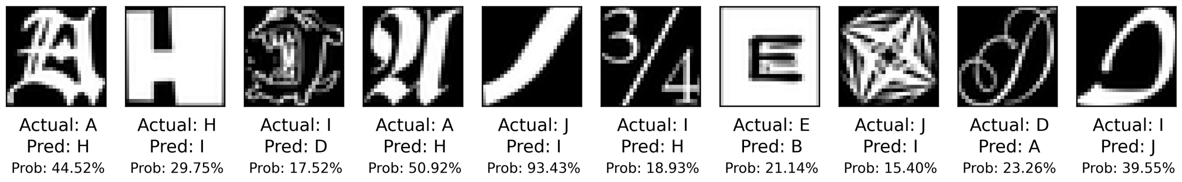


Figure 13: Randomly selected misclassified images on the notMNIST's test set.

3. Research: Adam Optimization Algorithm

The learning rate α is one of the most important hyper-parameters in artificial neural networks, responsible for the magnitude of change with relation to the gradients g in every update of the model's parameters θ. The most basic implementation of gradient descent, assumes that α is manually set before training and remains constant at every iteration of the algorithm. Additionally, there isn't a *rule of thumb* that tells us the optimal value given a certain model and dataset. Thus, it's crucial to find an appropriate one that minimizes the time required to train an algorithm whilst achieving a reasonably good performance. This basic approach is presented below mathematically:

$$\theta_{t+1} = \theta_t - \alpha \cdot g_t$$

There are two main issues with this strategy. Firstly (i), the optimality of α can vary during training and we're not able to change it. Secondly (ii), all the weights of the network are sharing a unique hyper-parameter that might not always fit them. For the first one, we could easily use a mathematical function that sets an initial large value at the start and progressively reduces it as the network reaches the global minimum. For the latest, however, it would be unfeasible to decide or iteratively fine-tune thousands or even millions of learning rates for each of the parameters of the network. Adam [1] is an adaptative learning rate optimization algorithm that tries to assess these two issues in an online manner. It merges into a single optimizer recent contributions to this field that yielded a better performance than using a single learning rate. In particular, this optimizer is greatly inspired by AdaGrad [2] and RMSProp [3].

AdaGrad was proposed as an algorithm that enables *personalized* learning rates for each weight, and also deals with changes in their magnitude during the optimization. It does so by scaling α according to the parameters' cumulative sum of squared gradients. In other words, weights that have been updated regularly in the past will gradually get lower learning rates, while those that didn't have much importance can still receive large updates at an advanced stage of the optimization. The algorithm makes this possible as it's storing the sum of squared gradients internally. The following equation shows its update rule:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\varepsilon + v_t^\theta}} \cdot g_t$$

where v_t^θ is calculated as $v_t^\theta = v_{t-1}^\theta + g_t^2$ and represents the accumulated sum of squared gradients at time t, and ε is just a small number to avoid diving by zero. The authors of the algorithm explained that the square root is added to control the magnitude of the sum of the squared gradients, which can eventually lead to very large values. This approach is much beneficial for datasets with sparse data, where some of the feature values do not show up frequently during training e.g. a rare but important pattern in the corner of an image. However, one of its major drawbacks is the lack of control over the decay of the learning rate, which is solely determined by an accumulated sum that will eventually prevent the neural network from learning.

RMSProp is an alternative to AdaGrad aimed at overcoming the issue we just mentioned. Instead of adding the squared gradients over time, it uses an Exponential Moving Average (EMA) that is biassed towards the most recent values of the squared gradients and determines the value of v_t^θ. Despite its simplicity, this approach keeps the algorithm learning even after a large number of iterations. As AdaGrads's update rule is shared with RMSProp, we show below the calculation of the Exponential Moving Average v_t^θ, where β is a parameter that acts as a weight for historical and current gradients.

$$v_t^\theta = \beta \cdot v_{t-1}^\theta + (1 - \beta) \cdot g_t^2$$

Finally, Adam is built using the properties that made these last two algorithms successful and becomes one of the most popular and widely used optimizers in Deep Learning. It takes from AdaGrad the root squared scaling factor of the learning rate, and from RMSProp the usage of Exponential Moving Averages. In particular, Adam calculates two EMAs using different weights β_1 and β_2, for g_t and g_t^2, respectively. Additionally, a bias correction is applied to the resulting values adjusting potential large updates at the beginning of the optimization. Indeed, this is an expected behavior of Exponential Moving Averages - the very first values are heavily biased towards a small sample of historical data, which is zero in terms of gradient calculations. These first steps of the algorithm are described and further explained below:

$$m_t^\theta = \beta_1 \cdot m_{t-1}^\theta + (1-\beta_1) \cdot g_t$$
$$v_t^\theta = \beta_2 \cdot v_{t-1}^\theta + (1-\beta_2) \cdot g_t^2$$

Essentially, m_t^θ is an EMA of the gradients, and v_t^θ is an EMA of the squared gradients as in RMSProp. The Authors of the algorithm propose default values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Thus, for the first iteration when $m_{t-1}^\theta = 0$ and $v_{t-1}^\theta = 0$, m_t^θ and v_t^θ are solely based on on a small portion of the current gradient determined by $(1-\beta_i)$, which is a small value. In order to overcome this issue, they introduce bias corrections that consist in scaling the EMAs to reduce the impact of the β_i parameters during the very first steps (note that RMSProp doesn't perform this correction):

$$\hat{m}_t^\theta = \frac{m_t^\theta}{(1-\beta_1)} \quad \hat{v}_t^\theta = \frac{v_t^\theta}{(1-\beta_2)}$$

Bias-corrected values are then used to adjust the parameters of the network using the update rule:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t^\theta}{\sqrt{\hat{v}_t^\theta} + \varepsilon}$$

Although the expression has been rearranged, Adam actually applies very few changes to AdaGrad and RMSProp's versions: (i) It includes the bias-corrected value of an EMA of the gradients $\hat{m}_t^\theta$ instead of the actual gradients g_t. (ii) The EMA representing the squared gradients is now unbiased $\hat{v}_t^\theta$. (iii) The scaling factor in the denominator no longer performs the squared root of ε and it's added separately for numerical stability - the Authors recommend setting a value of 1e$-$8 for it.

Empirical results have shown that Adam outperforms other adaptative techniques in a variety of convex and non-convex optimization problems using gradient-based models. It doesn't only reduce time spent searching for an appropriate learning rate, but also overcomes some of the issues found in popular algorithmic methods that manage this hyper-parameter during training. Ultimately, its computational efficiency and ability to deal with large feature data makes it ideal for deep neural architectures that usually face complex problems with sparse gradients and noisy objective functions.

References

[**1**] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization.”, *International Conference on Learning Representations*, 2014.

[**2**] J. C. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.”, *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, 2011.

[**3**] Tieleman, T. and Hinton, “G. Lecture 6.5 - RMSProp”, *COURSERA: Neural Networks for Machine Learning.*, Technical report, 2012.