UNIVERSITÀ DEGLI STUDI DI MILANO

Statistical Learning, Deep Learning and Artificial Intelligence

Instructor: Prof. Silvia Salini

Student: Nallapaneni Aditya Sai (933570) - Data Science and Economics

Project: SVHN dataset Classification using Deep Learning

Index:

1.Abstract

This report is based on Street View House Numbers(SVHN) dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. SVHN is obtained from house numbers in Google Street View images. This report describes the usage of neural networks for the classification task and we will see the comparison over the Adaptive Gradient Descent method and Stochastic gradient descent method by using Convolutional neural networks (CNN) in SVHN dataset.

2. The SVHN dataset, Problem Statement, Project Goal:

The Street View House Numbers (SVHN) is a real-world image dataset used for developing machine learning and object recognition algorithms. It is one of the commonly used benchmark datasets as It requires minimal data preprocessing and formatting. Although it shares some similarities with MNIST where the images are of small cropped digits, SVHN incorporates an order of magnitude more labeled data (over 600,000 digit images). It also comes from a significantly harder real world problem of recognizing digits and numbers in natural scene images. The images lack any contrast normalization, contain overlapping digits and distracting features which makes it a much more difficult problem compared to MNIST.

• 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10. • The dataset consists of 73,257 digits for training and 26,032 digits for testing. It also comes with an additional 531,131 somewhat less difficult samples that can be used as extra training data. Comes in two formats: 1. Original images with character level bounding boxes.

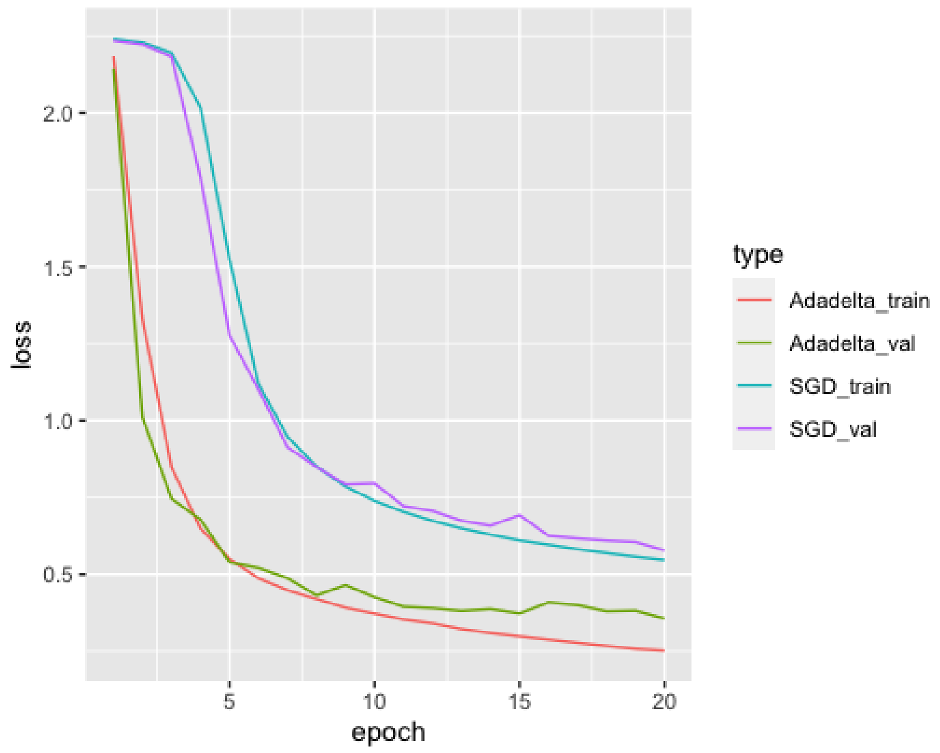
2. MNIST-like 32-by-32 images centered around a single character (many of the images do contain same distractors at the sides)

Format 2: Cropped Digits: Character level ground truth in an MNIST-like format. All digits have been resized to a fixed resolution of 32x32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32x32 pixels does not introduce aspect ratio distortions. Nevertheless this preprocessing introduces some distracting digits to the sides of the digit of interest. Loading the .mat files creates 2 variables: X which is a 4-D matrix containing the images, and y which is a vector of class labels. To access the images, X (: , : , : , I) gives the 32x32 RGB image, with class label y(i). The images contain grey levels centered in a 32*32*3 field. Sets of writers of the training set and test set were disjoint.

The goal of this project is comparison over Adaptive Gradient Descent method and Stochastic gradient descent method by using Convolutional neural networks (CNN).

3.Key Findings:

- CNN are good algorithms for Image Classification tasks
- Stochastic gradient descent (SGD) performs a parameter update for training label x(i) and y(i) whereas Adaptive Moment Estimation (Adam) computes adaptive learning rates for each parameter in storing an exponentially decaying average of past gradients.
- Adam is much faster than SGD, the default hyperparameters usually works better.
- Adam has convergence problems where SGD can converge better with longer training time.

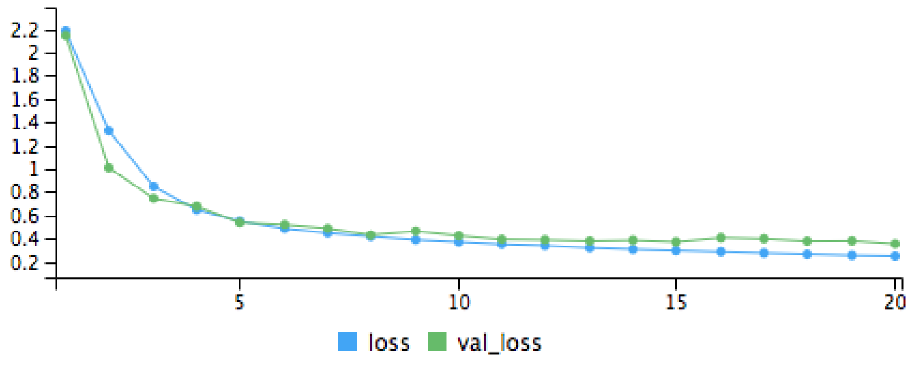


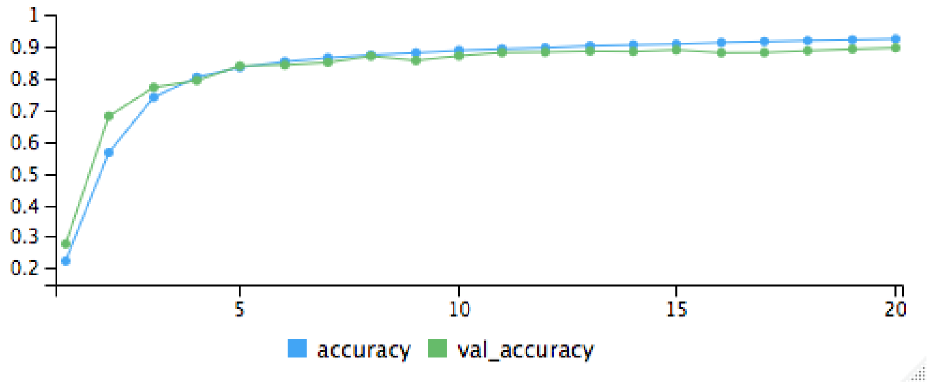
4. Analysis

4.1 CNN using Ada-delta:

Using a CNN, with 3 **Conv2D layers** (with 32, 32 and 64 filters respectively), 1 **MaxPool**.

Using **relu** activation function, with the final output layer containing 10 neurons (each corresponding to 1 different label) with a **SoftMax** activation.
SoftMax activation function for Classification as it creates an exponential probability distribution in the range of (0,1) making it easy for classification based on the activation with highest probability.
Loss function: Categorical Cross Entropy, which makes the labels mutually exclusive for each training data point making it suitable for our **Single-label Classification**
Trained the model using **Ada-delta** optimizer. With a batch size of 256 and 20% validation split, the model achieved a **val_accuracy** of **89.81%** for 20 epochs.

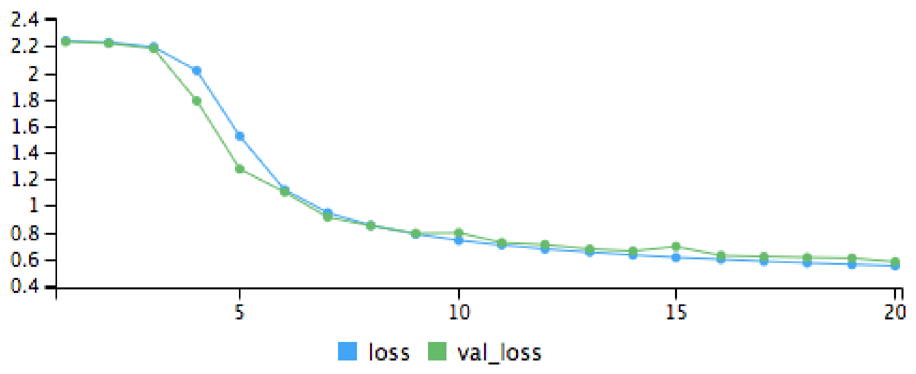


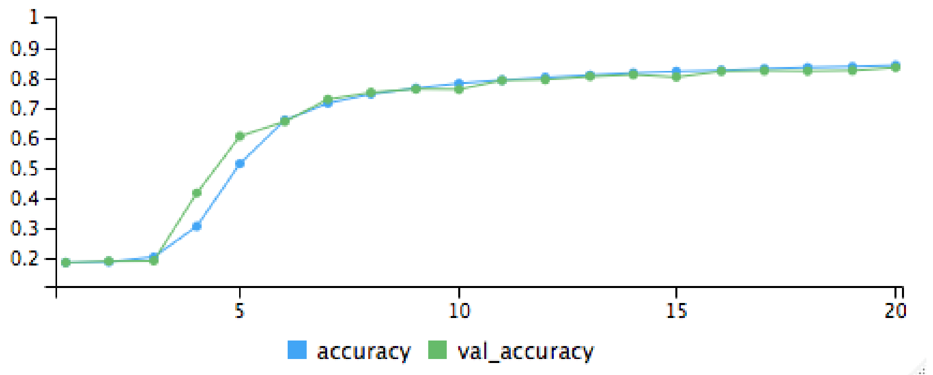


Achieved an accuracy of **90%** when evaluated on the test set.

4.2 CNN using SGD:

Using Stochastic Gradient Descent (SGD) optimizer with a less complex network arcitecture (using a less filter size of (3,3) instaed of (5,5), with 3 Conv2D and 2 MaxPool layers with 1 AvgPooling) to above, achieved an val_accuracy of **83.39%**
Also, please note the model was trained at a lower batch size of 64 so the weights updation is smooth and distributed.



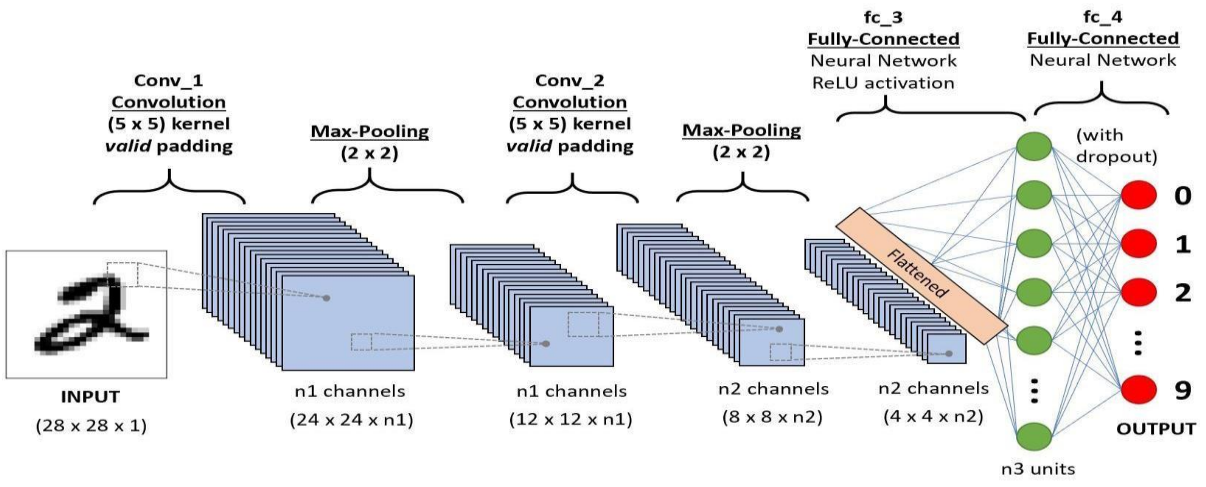


We see that SGD takes more training time to update the weights correctly when compared to an Adaptive Optimizer like ada-delta.
Acheived an accuracy of **84%** on test set when trained upto 20 epochs.

5. Theoretical background of methods used:

5.1 Convolutional Neural Network

Convolutional Neural Network (ConvNet or CNN) is a special type of Neural Network used effectively for image recognition and classification. They are highly proficient in areas like identification of objects, faces etc. CNN like any other Neural network learns through the process of Backpropagation. An Image is classified as a matrix of pixel values. Any image can be represented as a matrix made up of pixel values. Any component of an image is conventionally referred to as a Channel. Images obtained from a standard digital camera will consist of 3 channels – green, blue and red. These can be pictured as 3 two-dimensional matrices, one for each color, with pixel values between 0 to 255 and stacked over one another.



Grayscale image, which is an image with just one channel, we will use only grayscale images to have a single 2D matrix depicting an image. Pixel value range for the matrix is between 0 to 255, with 0 indicating black and 255 showcasing white.

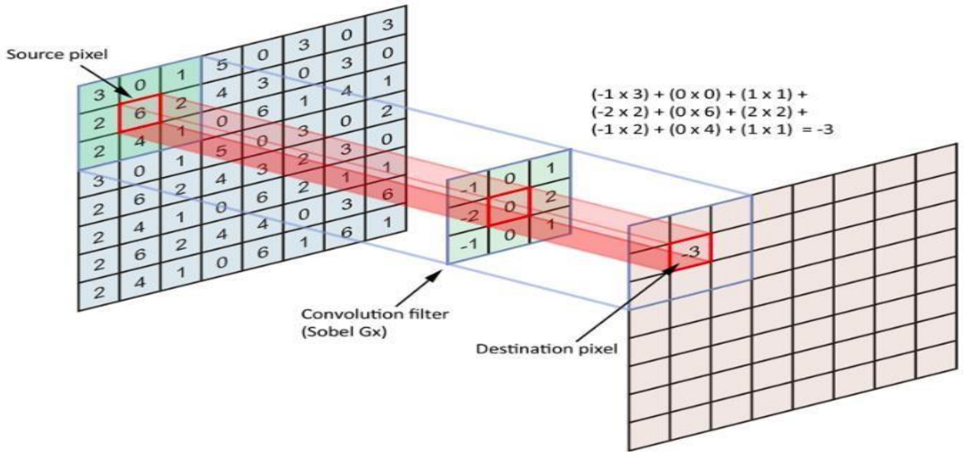
Convoluting:

The name ConvNets had been derived from an operator called 'Convolution'. The primary objective of this operator is the extraction of input image features. Convolution learns image features and works in coordination with pixels by using small squares of input data.

An image is composed on number of channels mentioned with the filter being applied on it, we can choose the number of filters. At end, we apply activation function; can be a Max function or **Re LU** function. As output, a tensor is created for each filter applied.

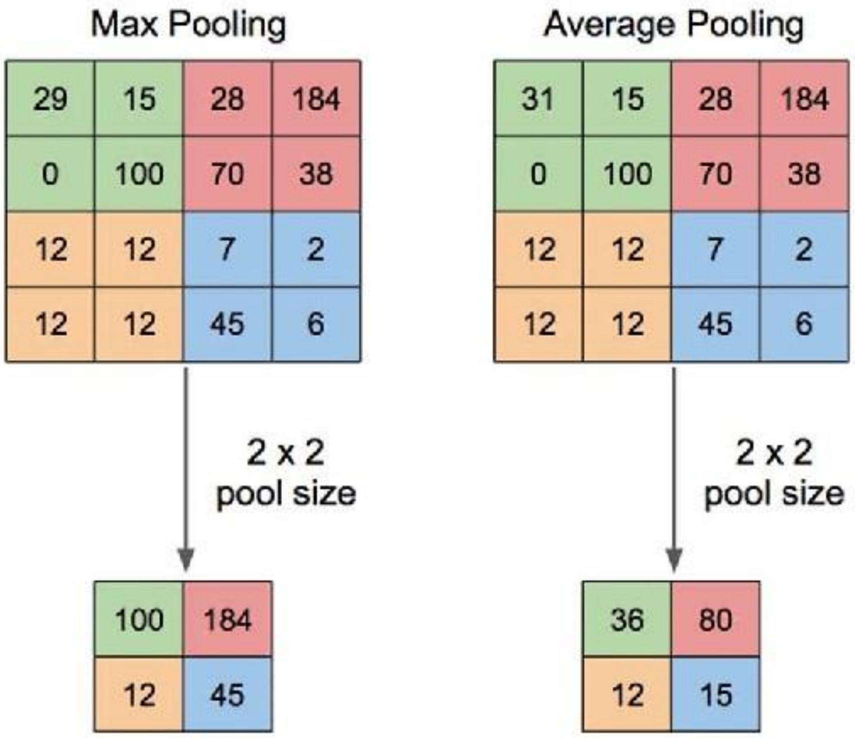
Again, these tensors are allowed as inputs to the next layer in the convolution. Then, number of filters has to be chosen and applied on the channels. Later, on applying activation function, the same Max function tensors are produced as output, equal to the number of filters applied earlier.

Re LU is an operation working element-wise i.e. is applied per pixel and substitutes every negative pixel value by 0 in the feature map. It serves the purpose of introducing non-linearity in ConvNet, because the maximum real-life data we will want to feed into our ConvNet is non-linear.


Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Pooling:

It aggregates some values into a single output value. It reduces the size of the output tensor from each input tensor. Depth remains the same after pooling. *Max Pooling* takes the maximum input of a particular convoluted feature. *Average Pooling* takes the average input of a particular convoluted feature. we first specify a spatial neighborhood (such as a 2×2 window) and then pick out the largest element of that feature rectified map within that window. Instead of largest element, if we pick the Average one it's called Average Pooling and when the summation of elements in the window is taken, we call it Sum Pooling.



Fully Connected:

This layer is described as a Multilayer Perceptron which utilizes a softmax activation function present in the output layer (we can use SVM as the classifier as well but we will limit out approach to softmax here). The words “Fully Connected”
in the fully connected layer indicate that every neuron in the next layer is connected to every individual neuron in the previous layer.Pooling and convolutional layer outputs depict high-resolution features of the input image. The fully connected layer on the basis of the training dataset utilizes these features for categorizing input images into different classes.

5.2 Back propagation in CNN

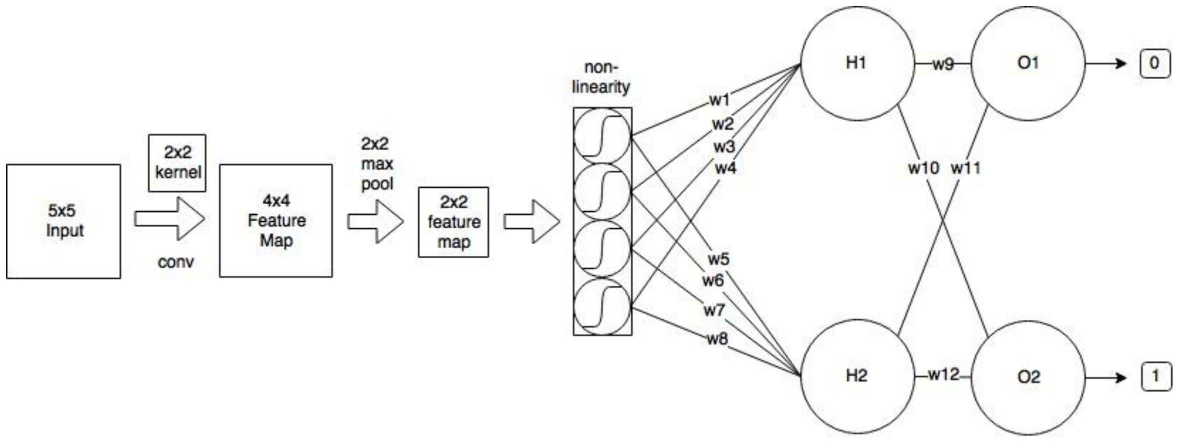
Convolutional layers are different they have a fixed number of weights governed by the choice of filter size and number of filters, but independent of the input size. The filter weights absolutely must be updated in backpropagation, since this is how they learn to recognize features of the input. Every layer in a neural net consists of

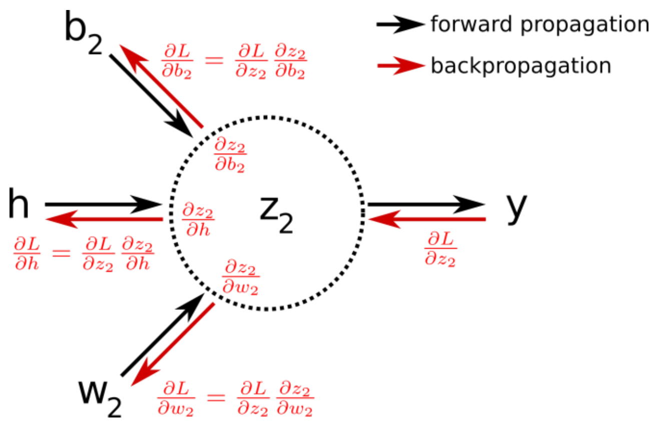
forward and backward computation, because of the backpropagation, Convolutional layer is one of the neural net layer
Each propagation involves the following steps:

- Propagation forward through the network to generate the output value(s)
- Calculation of the cost (error term)
- Propagation of the output activations back through the network using the training pattern target in order to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons.

For each weight, the following steps must be followed:

- The weight's output delta and input activation are multiplied to find the gradient of the weight.
- A ratio (percentage) of the weight's gradient is subtracted from the weight.




forward propagation
backpropagation
b_2
$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial b_2}$
$\frac{\partial z_2}{\partial b_2}$
h
$\frac{\partial z_2}{\partial h}$
z_2
y
$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial h}$
$\frac{\partial L}{\partial z_2}$
$\frac{\partial z_2}{\partial w_2}$
w_2
$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial w_2}$

Backpropagation is based around four fundamental equations. Together, those equations give us a way of computing both the error δl and the gradient of the cost function.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

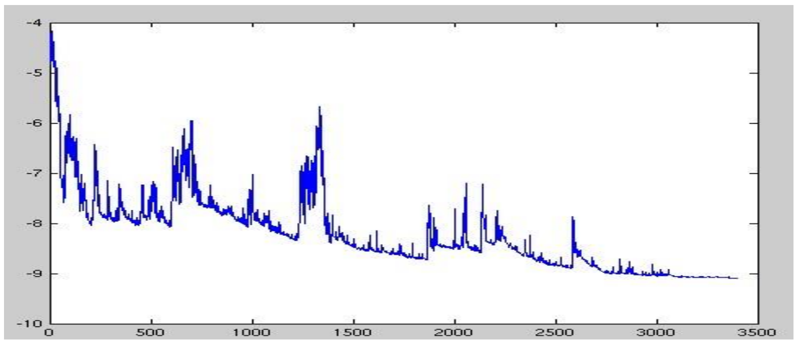
$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(BP4)}$$

5.3 Stochastic Gradient Descent(SGD):

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example x(i) and label y(i).

The use of SGD In the neural network setting is motivated by the high cost of running back propagation over the full training set. SGD can overcome this cost and still lead to fast convergence. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Image below.



Generally each parameter update in SGD is computed w.r.t a few training examples or a **minibatch**. The reason for this is, first this reduces the variance in the parameter update and can lead to more stable convergence, second this allows the computation to take advantage of highly optimized matrix operations that should be used in a well vectorized computation of the cost and gradient.

Mini batch of n training $\theta=\theta-\eta\cdot\nabla_\theta J(\theta;x^{(i:i+n)};y^{(i:i+n)})$.

A typical minibatch size is 256, although the optimal size of the minibatch can vary for different applications and architectures. In SGD the learning rate α is typically much smaller than a corresponding learning rate in batch gradient descent because there is much more variance in the update.

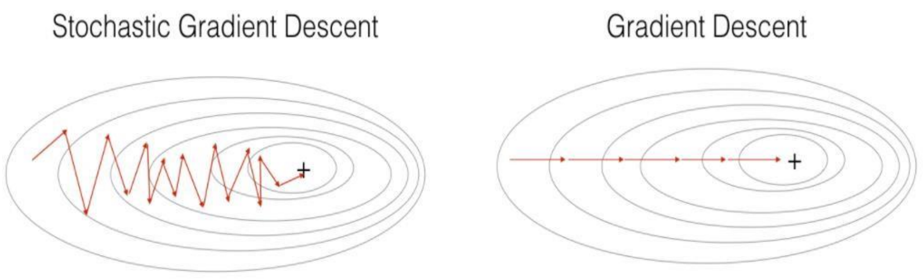
An better approach is to evaluate a held out set after each epoch and anneal the learning rate when the change in objective between epochs is below a small threshold. This tends to give good convergence to a local optima. If the data is given in some meaningful order, this can bias the gradient and lead to poor convergence. Generally a good method to avoid this is to randomly shuffle the data prior to each epoch of training

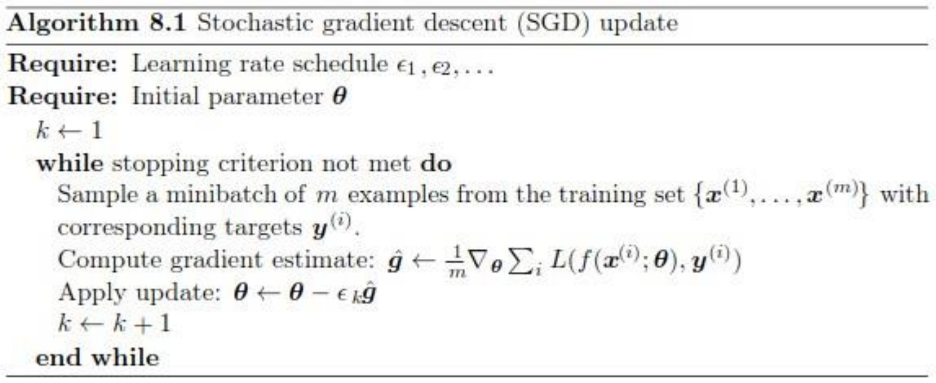
- Stochastic gradient methods can generally be written

$$w_{k+1} = w_k - \alpha_k \tilde{\nabla} f(w_k),$$

- Stochastic momentum methods are a second family of techniques that have been used to accelerate training which choose a local distance measure constructed using the entire sequence of iterates $(w_1,\cdots,w_k)$.. These methods can generally be written as

$$w_{k+1} = w_k - \alpha_k \nabla f(w_k + \gamma_k(w_k - w_{k-1})) + \beta_k(w_k - w_{k-1}).$$





Algorithm 8.1 Stochastic gradient descent (SGD) update

```
Require: Learning rate schedule $\epsilon_1, \epsilon_2, \dots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while
```

5.4 Adaptive Gradient Descent:

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. we use gradient descent to update the parameters of our model.

Ada delta optimization is a stochastic gradient descent method that is based on adaptive learning rate per dimension to address two drawbacks:

- The continual decay of learning rates throughout training
- The need for a manually selected global learning rate

Ada delta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients.

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments, Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.

6. Conclusions:

SVHN is a very large and extensive dataset that comes from a significantly more difficult problem where images contain a lot of clutter and noisy features. It seems to be under-utilized in the literature compared to MNIST, CIFAR-10 and CIFAR-100. Unlike MNIST and other datasets, preprocessing is common practice and very important for fairly comparing results. A form of contrast normalization, in particular local contrast normalization, is a common technique for preprocessing the SVHN dataset images. With regards to architecture, a convolutional architecture is quite common within the available benchmarks, which would be expected in a standard computer vision problem. Idea of Real time Data Augmentation may help the network achieve better performance – however, this will take training longer than usual.

7. Appendix: (R code)

```
 ## SVHN Dataset Classification using Artificial Neural Networks::


# 1 - Installing required packages::
devtools::install_github("rstudio/keras")  #skip if already
installed
library(keras)
install_keras()
library(tensorflow)
install_tensorflow(version = "nightly") #skip if already installed
library(keras)


require(R.matlab)
require(reticulate)


# 2 - Loading the dataset:: (remember to change below path to your
directory)
data_train <- readMat("/Users/adityanallapaneni/Downloads/svhn
datasets/train_32x32.mat")
data_test <- readMat("/Users/ adityanallapaneni/Downloads/svhn
datasets/test_32x32.mat")


x_train <- data_train$X
y_train <- data_train$y
```

```
x_test <- data_test$X
y_test <- data_test$y

# 3 - Preprocessing data suitable for CNN::
d <- dim(x_train)
train <- array(dim=c(d[4], d[1], d[2], d[3]))
for (i in 1:d[4])
{ train[i,,,] <- x_train[,,,i]
}

d = dim(x_test)
test = array(dim=c(d[4], d[1], d[2], d[3]))
for (i in 1:d[4])
{ test[i,,,] <- x_test[,,,i]
}

x_trainCNN <- array_reshape(train, c(dim(x_train)[4], 32, 32, 3))
x_testCNN <- array_reshape(test, c(dim(x_test)[4], 32, 32, 3))


# 3.3 - Rescaling Data::
x_trainCNN <- x_trainCNN / 255
x_testCNN <- x_testCNN / 255


# 3.3 - Organising the labels suitable for Classification::
y_train <- to_categorical(y_train-1, 10)
y_test <- to_categorical(y_test-1, 10)


## 4 - CNN architecture using Adaptive Gradient Descent::
```

```
model <- keras_model_sequential()
model %>%
  layer_conv_2d(filters = 32, kernel_size = c(5,5), activation =
'relu',
                input_shape = c(32, 32, 3), padding="same") %>%
  layer_max_pooling_2d(pool_size = c(3, 3)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(5,5), activation =
'relu', padding="same") %>%
  layer_average_pooling_2d(pool_size = c(3, 3)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(5,5), activation =
'relu', padding="same") %>%
  layer_max_pooling_2d(pool_size = c(3, 3)) %>%
  layer_flatten() %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)

model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)

history <- model %>%
  fit(
    x_trainCNN, y_train,
    epochs = 20,verbose = 1,
    batch_size = 256, validation_split = 0.2
  )

# evaluate accuracy on test set::
model %>% evaluate(x_testCNN, y_test, batch_size = 256, verbose =
1, sample_weight = NULL)
```

```
# prediction
model %>% predict_classes(x_testCNN)

## 5 - CNN Using SGD Optimizer::

modelSGD <- keras_model_sequential()
modelSGD %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation =
'relu',
                input_shape = c(32, 32, 3), padding="same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation =
'relu', padding="same") %>%
  layer_average_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation =
'relu', padding="same") %>%
  layer_max_pooling_2d(pool_size = c(3, 3)) %>%
  layer_flatten() %>%
  layer_dense(units = 10, activation = 'softmax')



modelSGD %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_sgd(),
  metrics = c('accuracy')
)

historySGD <- modelSGD %>%
  fit(
    x_trainCNN, y_train,
    epochs = 20,verbose = 1,
    batch_size = 64, validation_split = 0.2
```

```r
  )

# evaluate accuracy on test set::
modelSGD %>% evaluate(x_testCNN, y_test)

# prediction
modelSGD %>% predict_classes(x_testCNN)


##6. Comparing the 2 models::

library(tidyr)
library(tibble)
library(dplyr)
library(ggplot2)

compare_cx <- data.frame(
  Adadelta_train = history$metrics$loss,
  Adadelta_val = history$metrics$val_loss,

  SGD_train = historySGD$metrics$loss,
  SGD_val = historySGD$metrics$val_loss
) %>%
  rownames_to_column() %>%
  mutate(rowname = as.integer(rowname)) %>%
  gather(key = "type", value = "value", -rowname)

ggplot(compare_cx, aes(x = rowname, y = value, color = type)) +
  geom_line() +
  xlab("epoch") +
  ylab("loss")
```

8. References:

- https://ruder.io/optimizing-gradientdescent/index.html#gradientdescentvariants
- https://papers.nips.cc/paper/7003-the-marginal-value-of-adaptive-gradientmethods-in-machine-learning.pdf
- http://neuralnetworksanddeeplearning.com/chap2.html
- https://www.jefkine.com/general/2016/09/05/backpropagation-inconvolutional-neural-networks/