UNIVERSITÀ DEGLI STUDI
DI MILANO

STATISTICAL METHODS FOR MACHINE LEARNING

INSTRUCTOR/DOCENTE: Nicolò Cesa-Bianchi

Project: **Neural network classification with TensorFlow**

Dataset: SVHN (http://ufldl.stanford.edu/housenumbers/)

Student: Pramodh Chinthareddy (936461) - Data Science and Economics

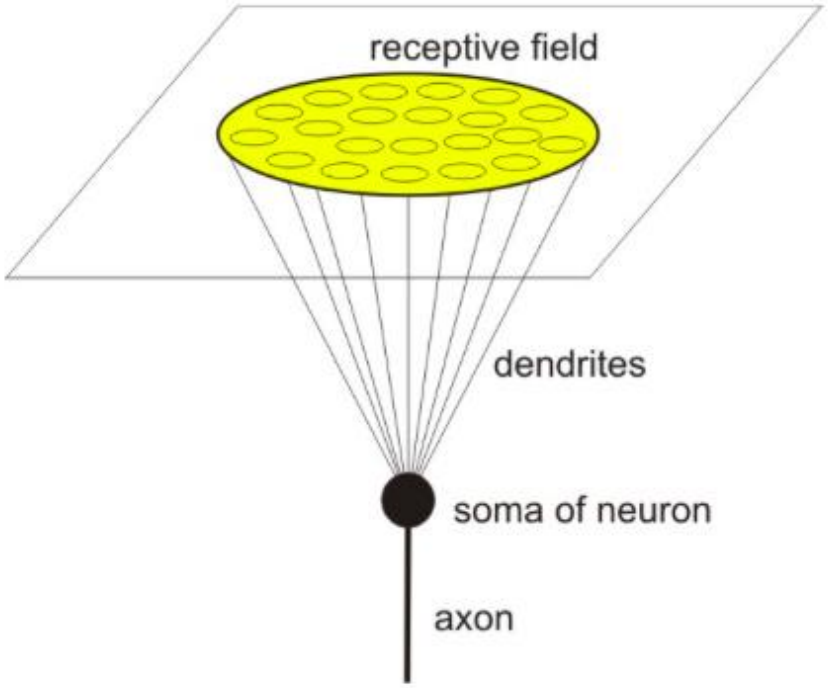
1.Convolutional Neural Network (CNN) :

In deep learning, a convolutional neural network (CNN) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as **shift invariant** or **space invariant artificial neural networks** (**SIANN**), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, medical image analysis and natural language processing.

The name “convolutional neural network” indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

CNNs are regularized versions of multilayer perceptrons. Multilayer perceptron's usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

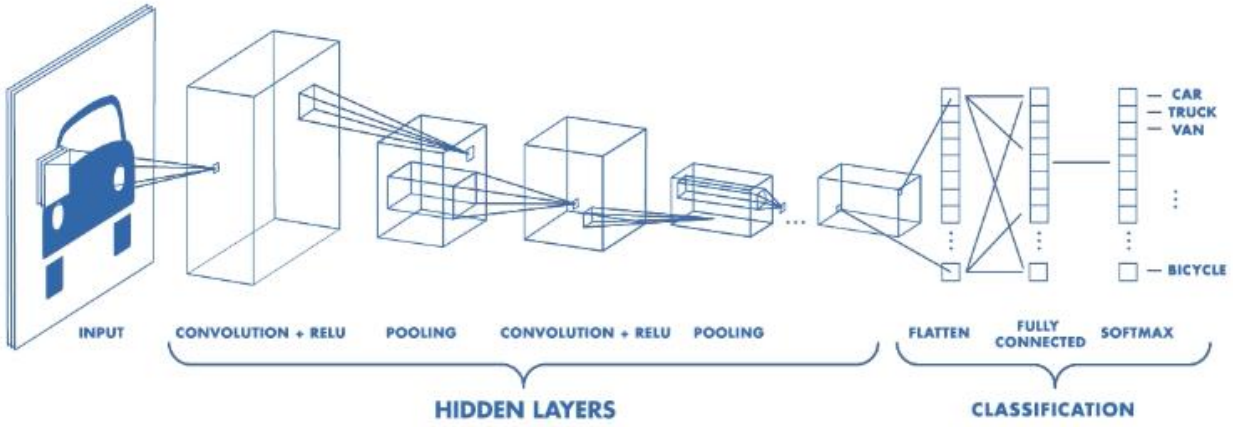


CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

1.1 Design:

A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that *convolve* with a multiplication or other dot product. The activation function is commonly a RELU layer, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution. The final convolution, in turn, often involves backpropagation in order to more accurately weight the end product.

Though the layers are colloquially referred to as convolutions, this is only by convention. Mathematically, it is technically a *sliding dot product* or cross-correlation. This has significance for the indices in the matrix, in that it affects how weight is determined at a specific index point.

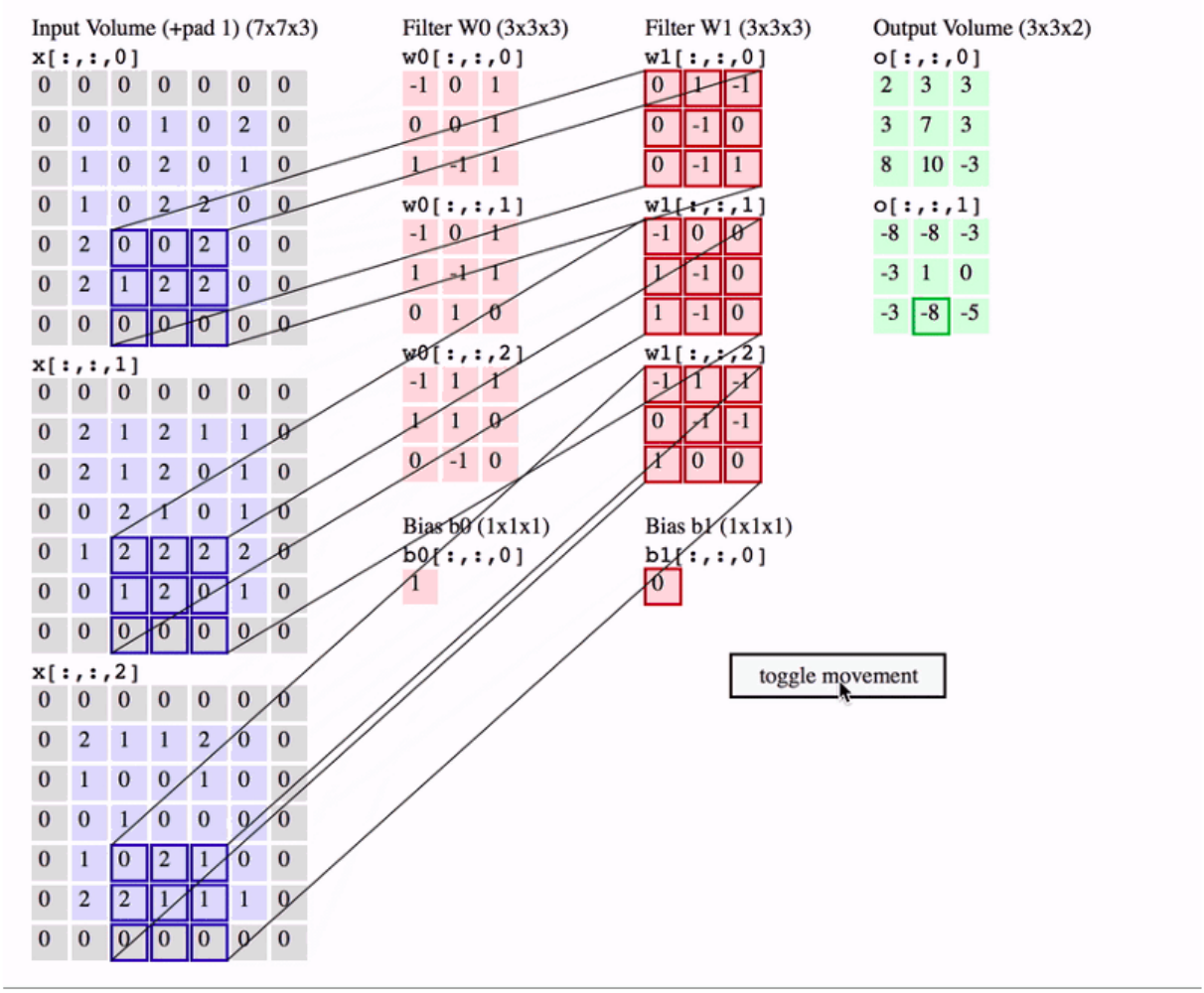


1.1.1 Convolutional:

When programming a CNN, the input is a 3D tensor with shape (number of images) x (image width) x (image height) x (image depth). Then after passing through a convolutional layer, the image becomes abstracted to a feature map, with shape (number of images) x (feature map width) x (feature map height) x (feature map channels). A convolutional layer within a neural network has the following attributes:

- Convolutional kernels defined by a width and height (hyper-parameters).
- The number of input channels and output channels (hyper-parameter).
- The depth of the Convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map.

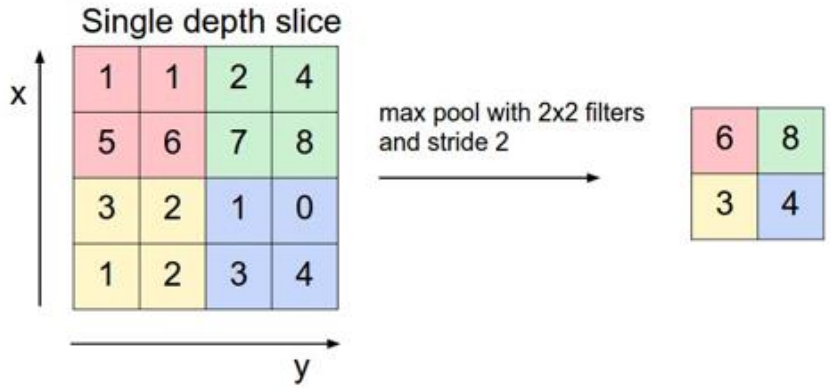
Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus. Each convolutional neuron processes data only for its receptive field. Although fully connected feedforward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10,000 weights for *each* neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5, each with the same shared weights, requires only 25 learnable parameters. In this way, it resolves the vanishing or exploding gradients problem in training traditional multi-layer neural networks with many layers by using backpropagation.



How convolution works with K = 2 filters, each with a spatial extent F = 3 , stride, S = 2, and input padding P = 1. — Source: http://cs231n.github.io/convolutional-networks/

1.1.2 Pooling:

Convolutional networks may include local or global pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2 x 2. Global pooling acts on all the neurons of the convolutional layer. We can use Max or Average pooling. *Max pooling* uses the maximum value from each of a cluster of neurons at the prior layer. *Average pooling* uses the average value from each of a cluster of neurons at the prior layer.



1.1.3 Fully Connected

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

1.1.4 Receptive Field:

In neural networks, each neuron receives input from some number of locations in the previous layer. In a fully connected layer, each neuron receives input from *every* element of the previous layer. In a convolutional layer, neurons receive input from only a restricted subarea of the previous layer. Typically, the subarea is of a square shape (e.g., size 5 by 5). The input area of a neuron is called its *receptive field*. So, in a fully connected layer, the receptive field is the entire previous layer. In a convolutional layer, the receptive area is smaller than the entire previous layer.

1.1.5 Weights:

Each neuron in a neural network computes an output value by applying a specific function to the input values coming from the receptive field in the previous layer. The function that is applied to the input values is determined by a vector of weights and a bias (typically real numbers). Learning, in a neural network, progresses by making iterative adjustments to these biases and weights.

The vector of weights and the bias are called *filters* and represent particular features of the input (e.g., a particular shape). A distinguishing feature of CNNs is that many neurons can share the same filter. This reduces memory footprint because a single bias and a single vector of weights are used across all receptive fields sharing that filter, as opposed to each receptive field having its own bias and vector weighting.

1.2 Distinguishing features:

CNN models mitigate the challenges posed by the MLP architecture by exploiting the strong spatially local correlation present in natural images. As opposed to MLPs, CNNs have the following distinguishing features:

- **3D volumes of neurons**. The layers of a CNN have neurons arranged in 3 dimensions: width, height and depth. where each neuron inside a convolutional layer is connected to only a small region of the layer before it, called a receptive field. Distinct types of layers, both locally and completely connected, are stacked to form a CNN architecture.
- **Local connectivity**: following the concept of receptive fields, CNNs exploit spatial locality by enforcing a local connectivity pattern between neurons of adjacent layers. The architecture thus ensures that the learned "filters" produce the strongest response to a spatially local input pattern. Stacking many such layers leads to non-linear filters that become increasingly global (i.e. responsive to a larger region of pixel space) so that the network first creates representations of small parts of the input, then from them assembles representations of larger areas.
- **Shared weights**: In CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map. This means that all the neurons in a given convolutional layer respond to the same feature within their specific response field. Replicating units in this way allows for the resulting feature map to be equivariant under changes in the locations of input features in the visual field, i.e. they grant translational equivariance.
- **Pooling:** In a CNN's pooling layers, feature maps are divided into rectangular sub-regions, and the features in each rectangle are independently down-sampled to a single value, commonly by taking their average or maximum value. In addition to reducing the sizes of feature maps, the pooling operation grants a degree of translational invariance to the features contained therein, allowing the CNN to be more robust to variations in their positions.

Together, these properties allow CNNs to achieve better generalization on vision problems. Weight sharing dramatically reduces the number of free parameters learned, thus lowering the memory requirements for running the network and allowing the training of larger, more powerful networks.

1.3 Building blocks:

A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume (e.g. holding the class scores) through a differentiable function. A few distinct types of layers are commonly used.

1.3.1 Convolutional layer:

Is the core building block of a CNN. This layer's parameters consist of a set of learnable **filters** (or **kernels**), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is **convolved** across the width and height of the

input volume, computing the **dot product** between the entries of the filter and the input and producing a 2-dimensional **activation map** of that filter. As a result, the network learns filters that activate when it detects some specific type of **feature** at some spatial position in the input.

Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

1.3.1.1 Local connectivity:

When dealing with high- dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume because such a network architecture does not take the spatial structure of the data into account. Convolutional networks exploit spatially local correlation by enforcing a **sparse local connectivit**y pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume.

The extent of this connectivity is a **hyperparameter** called the **receptive field** of the neuron. The connections are local in space (along width and height), but always extend along the entire depth of the input volume. Such an architecture ensures that the learnt filters produce the strongest response to a spatially local input pattern.

1.3.1.2 Spatial arrangement:

Three hyperparameters control the size of the output of the convolutional layer: the depth, stride and zero-padding.

- The *depth* of the output volume controls the number of neurons in a layer that connect to the same region of the input volume. These neurons learn to activate for different features in the input. For example, if the first convolutional layer takes the raw image as input, then different neurons along the depth dimension may activate in the presence of various oriented edges, or blobs of color.
- *Stride* controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1 then we move the filters one pixel at a time. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. When the stride is 2 then the filters jump 2 pixels at a time as they slide around. Similarly, for any integer **S > 0**, a stride of *S* causes the filter to be translated by *S* units at a time per output. In practice, stride lengths of **S >= 3** are rare. The receptive fields overlap less and the resulting output volume has smaller spatial dimensions when stride length is increased.
- Sometimes it is convenient to pad the input with zeros on the border of the input volume. The size of this padding is a third hyperparameter. Padding provides control of the output volume spatial size. In particular, sometimes it is desirable to exactly preserve the spatial size of the input volume.

The spatial size of the output volume can be computed as a function of the input volume size W, the kernel field size of the convolutional layer neurons K, the stride with which they are

applied S, and the amount of zero padding P used on the border. The formula for calculating how many neurons "fit" in a given volume is given by

[(W – K + 2P) / S] + 1

If this number is not an integer, then the strides are incorrect and the neurons cannot be tiled to fit across the input volume in a symmetric way. In general, setting zero padding to be **P = (K – 1) / 2** when the stride is **S = 1** ensures that the input volume and output volume will have the same size spatially. However, it's not always completely necessary to use all of the neurons of the previous layer. For example, a neural network designer may decide to use just a portion of padding.

1.3.1.3 Parameter sharing:

A parameter sharing scheme is used in convolutional layers to control the number of free parameters. It relies on the assumption that if a patch feature is useful to compute at some spatial position, then it should also be useful to compute at other positions. Denoting a single 2-dimensional slice of depth as a *depth slice*, the neurons in each depth slice are constrained to use the same weights and bias.

Since all neurons in a single depth slice share the same parameters, the forward pass in each depth slice of the convolutional layer can be computed as a convolution of the neuron's weights with the input volume. Therefore, it is common to refer to the sets of weights as a filter (or a **kernel**), which is convolved with the input. The result of this convolution is an **activation map**, a nd the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume. Parameter sharing contributes to the **translation invarinace** of the CNN architecture.

Sometimes, the parameter sharing assumption may not make sense. This is especially the case when the input images to a CNN have some specific centered structure; for which we expect completely different features to be learned on different spatial locations. One practical example is when the inputs are faces that have been centered in the image: we might expect different eye-specific or hair-specific features to be learned in different parts of the image. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a "locally connected layer".

1.3.2 Pooling layer:

Another important concept of CNNs is pooling, which is a form of non-linear **down-sampling**. There are several non-linear functions to implement pooling among which ***max pooling*** is the most common. It **partitions** the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum.

Intuitively, the exact location of a feature is less important than its rough location relative to other features. This is the idea behind the use of pooling in convolutional neural networks. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters, **memory footprint** and amount of computation in the network, and hence

to also control **overfitting**. It is common to periodically insert a pooling layer between successive convolutional layers in a CNN architecture. The pooling operation provides another form of translation invariance.

The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples at every depth slice in the input by 2 along both width and height, discarding 75% of the activations.

1.3.3 ReLU layer:

ReLU is the abbreviation of **rectified linear unit**, which applies the non-saturating activation function

f(x) = max (0, x), It effectively removes negative values from an activation map by setting them to zero. t increases the **nonlinear properties** of the **decision function** and of the overall network without affecting the receptive fields of the convolution layer.

ReLU is often preferred to other functions because it trains the neural network several times faster without a significant penalty to **generalization** accuracy.

1.3.4 Fully connected layer:

Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular (non-convolutional) **artificial neural networks**. Their activations can thus be computed as an **affine transforation**, with **matrix multiplication** followed by a bias offset (**vector addition** of a learned or fixed bias term).

1.3.5 Loss layer:

The "loss layer" specifies how **training** penalizes the deviation between the predicted (output) and **true** labels and is normally the final layer of a neural network.

1.4 Choosing hyperparameters:

1.4.1 Number of filters: Since feature map size decreases with depth, layers near the input layer will tend to have fewer filters while higher layers can have more. To equalize computation at each layer, the product of feature values v_a with pixel position is kept roughly constant across layers. Preserving more information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) non-decreasing from one layer to the next.

The number of feature maps directly controls the capacity and depends on the number of available examples and task complexity.

1.4.2 Filter shape:

Common filter shapes found in the literature vary greatly, and are usually chosen based on the dataset.

The challenge is, thus, to find the right level of granularity so as to create abstractions at the proper scale, given a particular dataset, and without **overfitting**.

1.4.3 Max pooling shape:

Typical values are 2×2. Very large input volumes may warrant 4×4 pooling in the lower layers. However, choosing larger shapes will dramatically **reduce the dimension** of the signal, and may result in excess **information loss**. Often, non-overlapping pooling windows perform best.

1.5 Regularization methods:

Is a technique to prevent **overfitting**. CNNs use various types of regularization.

1.5.1 Empirical: Dropout, DropConnect, Stochastic pooling, Artificial data

1.5.2 Explicit: Early Stopping, Number of parameters, Weight decay, Max norm constraints

2. The Street View House Numbers (SVHN) Dataset:

Much similar to MNIST(images of cropped digits), but SVHN contains much more labeled data (over **600,000 images**) with real world problems of recognizing digits and numbers in **natural scene images.**

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images.

2.1 Overview:

- 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10.
- 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use 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 some distractors at the sides).

I used the Format 2 version of the dataset available at (http://ufldl.stanford.edu/housenumbers/)

Format 2: Cropped Digits: train_32x32.mat, test_32x32.mat , extra_32x32.mat

Character level ground truth in an MNIST-like format. All digits have been resized to a fixed resolution of 32-by-32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32-by-32 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 i-th 32-by-32 RGB image, with class label y(i).

3. Description of Experiments:

A simple **Convolutional neural network** to classify SVHN images. I used the Keras Sequential API. Used the loadmat library from scipy.io to load the Format 2 version (.mat files) of the dataset. Separated the Images and the labels from both the train_32x32.mat and test_32x32.mat into **X_train** , **y_train**, **X_test**, and **y_test** respectively.

Here we are classifying 32 x 32 cropped images given in format 2 using CNN architecture.

3.1 Data preprocessing:

Pre-processing the 32-by-32 images from the SVHN dataset centered around a single digit. In this dataset all digits have been resized to a fixed resolution of 32-by-32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32-by-32 pixels does not introduce aspect ratio distortions

Transposing the the train and test data by converting it from:

(width, height, channels, size) -> (size, width, height, channels)

Normalized the pixel values to be between 0 and 1

Verifying the data: To verify that the dataset looks correct, defined a function to plot some images from the training set, test set and display the class name below each image.

Converted the Label 10's to 0's

3.2.1 Network Architecture:

Created the convolutional base: The 6 lines of code below define the convolutional base using a common pattern: a stack of Conv2D and MaxPooling2Dlayers.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. If you are new to these dimensions, color_channels refers to (R,G,B). In this example, you will configure our CNN to process inputs of shape (32, 32, 3), which is the format of SVHN images. You can do this by passing the argument input_shape to our first layer.

Displaying the architecture of our model so far:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 4, 4, 64)          36928
=================================================================
Total params: 56,320
Trainable params: 56,320
Non-trainable params: 0
_________________________________________________________________
```

Above, you can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, you can afford (computationally) to add more output channels in each Conv2D layer.

Add Dense layers on top: To complete our model, you will feed the last output tensor from the convolutional base (of shape (3, 3, 64)) into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. SVHN has 10 output classes, so you use a final Dense layer with 10 outputs and a softmax activation.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Below is the complete architecture of our model:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                65600
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

As you can see, our (3, 3, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

3.2.2 Compile and train the model:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_test, y_test))
```

```
Train on 73257 samples, validate on 26032 samples
Epoch 1/10
73257/73257 [==============================] - 236s 3ms/sample - loss: 0.8694 - acc: 0.7209 - val_loss: 0.5398 - val_
acc: 0.8444
Epoch 2/10
73257/73257 [==============================] - 219s 3ms/sample - loss: 0.4304 - acc: 0.8727 - val_loss: 0.4457 - val_
acc: 0.8719
Epoch 3/10
73257/73257 [==============================] - 209s 3ms/sample - loss: 0.3638 - acc: 0.8928 - val_loss: 0.3876 - val_
acc: 0.8885
Epoch 4/10
73257/73257 [==============================] - 220s 3ms/sample - loss: 0.3215 - acc: 0.9055 - val_loss: 0.4425 - val_
acc: 0.8691
Epoch 5/10
73257/73257 [==============================] - 206s 3ms/sample - loss: 0.2939 - acc: 0.9141 - val_loss: 0.3587 - val_
acc: 0.8993
Epoch 6/10
73257/73257 [==============================] - 189s 3ms/sample - loss: 0.2739 - acc: 0.9194 - val_loss: 0.3886 - val_
acc: 0.8906
Epoch 7/10
73257/73257 [==============================] - 182s 2ms/sample - loss: 0.2544 - acc: 0.9250 - val_loss: 0.3644 - val_
acc: 0.8984
Epoch 8/10
73257/73257 [==============================] - 180s 2ms/sample - loss: 0.2353 - acc: 0.9310 - val_loss: 0.3572 - val_
acc: 0.9022
Epoch 9/10
73257/73257 [==============================] - 172s 2ms/sample - loss: 0.2234 - acc: 0.9329 - val_loss: 0.3656 - val_
acc: 0.8997
Epoch 10/10
73257/73257 [==============================] - 181s 2ms/sample - loss: 0.2069 - acc: 0.9390 - val_loss: 0.3659 - val_
acc: 0.9022
```

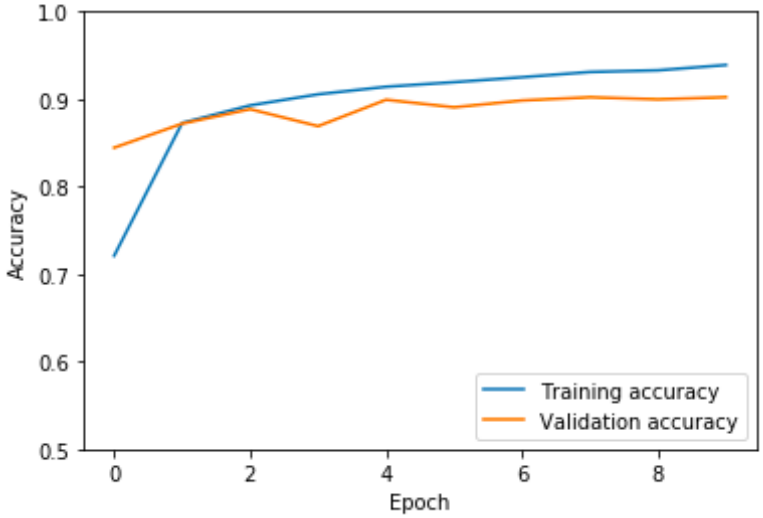
3.2.3 Evaluating the model:

In [26]:
```
# Training vs validation accuracy:

plt.plot(history.history['acc'], label='Training accuracy')
plt.plot(history.history['val_acc'], label = 'Validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(X_test,  y_test, verbose=2)
```

```
 - 14s - loss: 0.3659 - acc: 0.9022
```



In [27]:
```
print(test_acc)
```

```
0.9021973
```

Our simple CNN has achieved a 90% accuracy on the test set. We now explore other variants of network architecture to try improve the predictive performance.

3.3.1 Alternate Network Architecture 1:

In [22]:
```
from keras.layers import Dropout
model2 = models.Sequential()

model2.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model2.add(layers.Conv2D(64, (3, 3), activation='relu'))

model2.add(layers.MaxPooling2D((2, 2)))

model2.add(layers.Dropout(0.25))

model2.add(layers.Flatten())

model2.add(layers.Dense(128, activation='relu'))

model2.add(layers.Dropout(0.5))

model2.add(layers.Dense(10, activation='softmax'))

model2.compile(optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

history2 = model2.fit(X_train, y_train, epochs=30, batch_size = 256,
                      validation_data=(X_test, y_test))
```

We added one more Conv2D layer with 64 output channels before pooling, and used Dropout to avoid overfitting. We complete our model by feeding the last output tensor from the convolutional base (of shape (3, 3, 64)) into 2 Dense layers to perform classification after Flattening. Our final Dense layer with 10 outputs and a softmax activation. I trained the model a bit longer than previous model by using 30 epochs and also sent the images for training using a batch of 256 images at each pass.

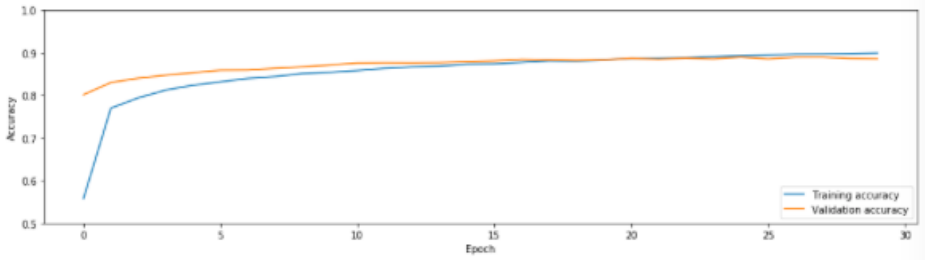
```
Epoch 29/30
73257/73257 [==============================] - 1817s 25ms/sample - l
oss: 0.3110 - accuracy: 0.8978 - val_loss: 0.4152 - val_accuracy: 0.
8867
Epoch 30/30
73257/73257 [==============================] - 274s 4ms/sample - los
s: 0.3053 - accuracy: 0.8994 - val_loss: 0.4165 - val_accuracy: 0.88
59
```

In [25]:
```
# Training vs validation accuracy:

plt.plot(history2.history['accuracy'], label='Training accuracy')
plt.plot(history2.history['val_accuracy'], label = 'Validation accurac
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model2.evaluate(X_test,  y_test, verbose=2)
```

```
26032/1 - 33s - loss: 0.3728 - accuracy: 0.8859
```



In [26]:
```
print(test_acc)
```

```
0.8859097
```

However, the accuracy didn't improve over our initial model.

3.4.1 Alternate Network Architecture 2:

Hence used the initial model as reference and tweaked it by adding more Conv2D with 32 and 64 input channels before 2 stages of pooling along with Dropout, and also implemented Batch Normalization by manually setting the hyperparameters before passing the output to the Dense layer for Classification. I used "relu" activation function for Batch Normalization (Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1)

Further, preprocessed the image (real-time data augmentation) for better performance of our model by applying ZCA whitening using ImageDataGenerator and finally fitted the model with 256 as batch_size and trained for 40 epochs to achieve an accuracy of 95% on the test set:

In [40]:

```
model3 = models.Sequential()

model3.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model3.add(layers.Conv2D(32, (3, 3), activation='relu'))

model3.add(layers.MaxPooling2D(pool_size=(2, 2)))

model3.add(layers.Conv2D(64, (3, 3), activation='relu'))
model3.add(layers.Conv2D(64, (3, 3), activation='relu'))

model3.add(layers.MaxPooling2D(pool_size=(2, 2)))

model3.add(layers.Dropout(0.2))
model3.add(layers.Flatten())

model3.add(layers.BatchNormalization(epsilon=0.001, axis=-1, momentum=0.99, weights=None, beta_initializer='zero',  gar
model3.add(layers.Activation('relu'))
model3.add(layers.BatchNormalization(epsilon=0.001, axis=-1, momentum=0.99, weights=None, beta_initializer='zero',  gar
model3.add(layers.Activation('relu'))

model3.add(layers.Dense(1024))
model3.add(layers.Activation('relu'))
model3.add(layers.Dropout(0.2))
model3.add(layers.Dense(10))
model3.add(layers.Activation('softmax'))

model3.compile(loss='sparse_categorical_crossentropy',
              optimizer="adam",
              metrics=['accuracy'])


datagen = ImageDataGenerator(
featurewise_center=False,  # Set input mean to 0 over the dataset
samplewise_center=False,  # Set each sample mean to 0
featurewise_std_normalization=False,  # Divide inputs by std of the dataset
samplewise_std_normalization=False,  # Divide each input by its std
zca_whitening=False,  # Apply ZCA whitening
rotation_range=5,  # Randomly rotate images in the range (0 to 180 degrees)
width_shift_range=0.1,  # Randomly shift images horizontally (fraction of total width)
height_shift_range=0.1,  # Randomly shift images vertically (fraction of total height)
horizontal_flip=False,  # Randomly flip images
vertical_flip=False)  # Randomly flip images
```

```
# Compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied).
datagen.fit(X_train)

# Fit the model on the batches generated by datagen.flow().
history3 = model3.fit_generator(datagen.flow(X_train, y_train,
                                 batch_size=256), steps_per_epoch=None, epochs=40,
                                 validation_data=(X_test, y_test))
```

```
curacy: 0.9391
Epoch 35/40
287/287 [==============================] - 655s 2s/step - loss: 0.1949 - accuracy: 0.9429 - val_loss: 0.2035 - val_ac
curacy: 0.9469
Epoch 36/40
287/287 [==============================] - 678s 2s/step - loss: 0.1886 - accuracy: 0.9439 - val_loss: 0.2046 - val_ac
curacy: 0.9464
Epoch 37/40
287/287 [==============================] - 644s 2s/step - loss: 0.1887 - accuracy: 0.9436 - val_loss: 0.2187 - val_ac
curacy: 0.9435
Epoch 38/40
287/287 [==============================] - 4044s 14s/step - loss: 0.1934 - accuracy: 0.9425 - val_loss: 0.2146 - val_
accuracy: 0.9448
Epoch 39/40
287/287 [==============================] - 647s 2s/step - loss: 0.1866 - accuracy: 0.9436 - val_loss: 0.1958 - val_ac
curacy: 0.9501
Epoch 40/40
287/287 [==============================] - 665s 2s/step - loss: 0.1854 - accuracy: 0.9437 - val_loss: 0.1955 - val_ac
curacy: 0.9504
```

In [41]:
```
# Training vs validation accuracy:

plt.plot(history3.history['accuracy'], label='Training accuracy')
plt.plot(history3.history['val_accuracy'], label = 'Validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model3.evaluate(X_test,  y_test, verbose=2)
```

```
26032/1 - 41s - loss: 0.1046 - accuracy: 0.9504
```

4. Final comments on Experiments:

Our simple CNN has achieved a test accuracy of over 90%. And the 2 other models gave an accuracy of 88% and 95% respectively. We see that Batch Normalization and Augmenting the images has significantly increased the performance of our model.

While the state-of-the-art results currently stand at 1.69% test error when used a Generalized Pooling (https://arxiv.org/pdf/1509.08985.pdf). This can be understood over the fact we have only used the train data and not the extra data owing to the computation capacity, and implemented a basic CNN architecture.

References:

[1] https://en.wikipedia.org/wiki/Convolutional_neural_network

[2] http://ufldl.stanford.edu/housenumbers/

[3] https://www.tensorflow.org/tutorials/images/cnn

[4] https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/

[5] https://agi.io/2018/01/31/getting-started-street-view-house-numbers-svhn-dataset/

[6] https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization

[7] https://keras.io/preprocessing/image/