

What is the linear projection W operator in PCA? $Y = XW$

In Principal Component Analysis (PCA), the linear projection operator W is a matrix that is used to transform the original data (X) into a new set of uncorrelated variables (Y). The columns of the matrix W are the eigenvectors of the covariance matrix of X , and the rows of Y are the transformed variables, also known as the principal components. The transformation is such that the first principal component has the largest possible variance, and each subsequent component has the highest variance possible under the constraint of being orthogonal to the previous components.

What is Fully connected scalar product and how can I use it in a layer? Which are its parameters?

A fully connected scalar product, also known as a fully connected layer, is a type of neural network layer where every neuron in the current layer is connected to every neuron in the next layer. The scalar product is the dot product of the input vector and the weight matrix. In other words, it is a matrix multiplication between the input and the weight matrix followed by a bias term and an activation function.

In a neural network, a fully connected layer can be implemented using a library such as TensorFlow or PyTorch by instantiating a dense layer object and specifying the number of neurons in the layer. For example, in TensorFlow, it can be done as follows:

Copy code

```
from tensorflow.keras import layers
```

```
#create a fully connected layer with 32 neurons  
fc_layer = layers.Dense(32)
```

The parameters of a fully connected layer are the weight matrix and bias term. The weight matrix is learned during training and it represents the strength of the connection between the neurons in the current layer and the next layer. The bias term is also learned during training and is added to the output of the scalar product. The activation function is also a parameter that is specified when creating the layer and it can be any non-linear function such as ReLU, sigmoid, tanh, etc.

What is Bayesian Information Criterion

The Bayesian Information Criterion (BIC) is a model selection criterion that is used to compare different models and select the one that best explains the data. It is based on the principle of parsimony, which states that, given two models that fit the data equally well, the simpler one is preferred.

BIC is defined as: $BIC = k * \ln(n) - 2 * \ln(L)$,

where k is the number of parameters in the model, n is the number of observations, and L is the likelihood of the data given the model.

A lower BIC value indicates a better model. The idea behind BIC is that it penalizes models with more parameters, thus encouraging simpler models. The BIC criterion balances the fit of the model to the data with the complexity of the model. BIC can be used to compare different models, such as linear regression models with different numbers of predictors or different types of probability distributions, for example.

It is important to note that BIC is a criterion to compare models, it does not provide an absolute measure of goodness of fit, and it is based on the assumption that the data is generated by the same process as the model, also it is not always the best criterion to use in all situations depending on the complexity of the model and the size of the sample.

What is the confusion matrix in model selection?

A confusion matrix is a table that is used to evaluate the performance of a classification model. It is a way to represent the performance of a model by summarizing the number of correct and incorrect predictions made by the model for each class.

A confusion matrix typically contains four main elements: true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN).

True positives are the cases in which the model correctly predicted the positive class. False positives are the cases in which the model predicted the positive class but the true class was negative. True negatives are the cases in which the model correctly predicted the negative class and false negatives are the cases in which the model predicted the negative class but the true class was positive.

The confusion matrix is used to calculate several performance measures such as accuracy, precision, recall, and F1-score. These measures can be used to evaluate a model's performance, compare different models, and select the best one.

It is important to note that the model selection based on confusion matrix is only one aspect of the process, other aspects like overfitting, underfitting, computational cost, domain knowledge, etc. should also be considered. Also, if the dataset is imbalanced, the accuracy may not be a suitable metric, and other metrics such as precision, recall and f1-score should be used instead.

What is the best loss function for classifying ? Machine larning. Why?

The choice of a loss function for classification tasks depends on the specific problem and the type of model being used. Some common loss functions used for classification are:

Binary Cross-Entropy Loss (BCE): This loss function is used for binary classification problems where the output of the model is a probability between 0 and 1. It is defined as the negative log-likelihood of the true labels given the predicted probabilities.

Categorical Cross-Entropy Loss (CCE): This loss function is used for multi-class classification problems where the output of the model is a probability distribution over

multiple classes. It is defined as the negative log-likelihood of the true labels given the predicted probabilities.

Hinge Loss: This loss function is used for maximum-margin classification problems such as support vector machines (SVMs). It is defined as the sum of the margin violations for all misclassified examples.

Focal Loss: This loss function is used for classification problem where the dataset is imbalanced, and the goal is to balance the positive and negative samples. It is defined as the cross-entropy loss, but with a weighting term that focuses on the hard examples.

There is no one best loss function for all classification problems. The choice of a loss function depends on the specific problem, the type of model being used, and the desired properties of the solution. For example, if the dataset is imbalanced and the goal is to balance the positive and negative samples, the Focal Loss is a good choice. On the other hand, if the dataset is balanced and the goal is to obtain a high accuracy, the Cross-Entropy Loss may be a good choice.

It is important to test multiple loss functions and carefully evaluate the performance of the model using different evaluation metrics before making a final decision.

How can you explain that, in a standard classification problem, precision and recall are most often contradictory objectives (if you want to increase one you will probably decrease the other one, for a specific model)? Confusion Matrix

In a standard classification problem, precision and recall are often contradictory objectives because they measure different aspects of a model's performance.

Precision is the proportion of true positive predictions (i.e. the number of true positives) out of all positive predictions made by the model (i.e. the number of true positives plus the number of false positives). It measures the ability of the model to correctly identify positive examples. A model with high precision is able to correctly identify positive examples with a low rate of false positives.

Recall, on the other hand, is the proportion of true positive predictions (i.e. the number of true positives) out of all actual positive examples (i.e. the number of true positives plus the number of false negatives). It measures the ability of the model to identify all positive examples. A model with high recall is able to identify most of the positive examples, even if it also has a high rate of false positives.

In a confusion matrix, precision and recall are related to the true positive (TP) and false positive (FP) and false negative (FN) predictions. If the model is designed to increase precision, it will make fewer false positive predictions, but it may also miss some actual positive examples, resulting in a decrease in recall. Conversely, if the model is designed to

increase recall, it will make more true positive predictions, but it may also make more false positive predictions, resulting in a decrease in precision.

Therefore, precision and recall are often contradictory objectives in a classification problem, and the choice of a model that balances these objectives will depend on the specific problem and the desired trade-off between these two performance metrics. In some cases, it may be desirable to have a model with high precision (low false positive rate) and in other cases, a model with high recall (low false negative rate) would be more appropriate. It can also be useful to use F1-score which is a trade-off between precision and recall.

A kernel method (for example a Support Vector Machine - SVM) transforms the input features in a nonlinear way; the new features are then used in a standard classification algorithm (for example a large margin classifier in the SVM). It is easy to understand that working with new features will change the classification performances, and that if you are lucky with the nonlinear transformation of inputs, the performances can increase. But besides the "chance" to increase the performances, how can you explain that generally speaking, indeed the performances increase? What are the conditions that you have to satisfy (on the nonlinear transformation) in order to guarantee (again, generally speaking) that the performances will increase?

Kernel methods, such as support vector machines (SVMs), can improve classification performance by transforming the input features in a non-linear way. The transformed features, also known as kernel features, can capture complex non-linear relationships in the data that are not easily captured by linear methods.

The reason why kernel methods can generally improve performance is due to the ability of the kernel transformation to map the input data into a higher-dimensional space where linear decision boundaries can separate the classes more effectively. In other words, the kernel trick maps the data into a higher-dimensional space where it becomes linearly separable, even if it isn't in the original feature space.

In order to guarantee that the performance will generally increase when using kernel methods, there are a few conditions that need to be satisfied:

The kernel function should be chosen appropriately for the problem at hand. Different kernel functions such as the polynomial kernel, the radial basis function (RBF) kernel, and the sigmoid kernel are suitable for different types of data and different types of problems.

The regularization parameter C should be chosen appropriately. A large value of C will result in a decision boundary that is close to the training data, which may lead to overfitting. A small value of C will result in a decision boundary that is far from the training data, which may lead to underfitting.

The kernel scale parameter, if present in the kernel function, should be chosen appropriately. A large scale parameter will result in a kernel feature space that is less sensitive to small

variations in the input data, and a small scale parameter will result in a kernel feature space that is more sensitive to small variations in the input data.

Lastly, the number of support vectors should be chosen appropriately. A large number of support vectors will result in a decision boundary that is close to the training data, which may lead to overfitting. A small number of support vectors will result in a decision boundary that is far from the training data, which may lead to underfitting.

It is important to note that kernel methods are sensitive to the choice of kernel function, the regularization parameter, the kernel scale parameter, and the number of support vectors. Therefore, it is important to carefully tune these parameters in order to achieve optimal performance.

For a simple, linearly separable, binary classification problem, explain why you would prefer (or not) to use a large-margin classifier rather than a perceptron.

A large-margin classifier and a perceptron are both linear classifiers that can be used for simple, linearly separable, binary classification problems. However, they have different properties and may be preferred in different situations.

A large-margin classifier, such as a support vector machine (SVM), finds the decision boundary that maximizes the margin, which is the distance between the decision boundary and the closest training examples of each class. This results in a classifier that is more robust to noise and outliers in the training data, as it tries to find a decision boundary that is as far as possible from the training examples.

A perceptron, on the other hand, finds the decision boundary that separates the training examples with the smallest number of misclassifications. This results in a classifier that is more sensitive to noise and outliers in the training data, as it tries to find a decision boundary that is as close as possible to the training examples.

Therefore, a large-margin classifier would be preferred in situations where the training data contains noise and outliers, as it is more robust to these types of errors. On the other hand, if the training data is relatively clean, a perceptron would be preferred as it is computationally simpler and faster to train.

It is also worth noting that a large-margin classifier can be extended to non-linearly separable problems using kernel trick while a perceptron is only limited to linearly separable problems.

In summary, a large-margin classifier, like SVM, is more robust to noise and outliers in the training data, while a perceptron is more sensitive to noise and outliers. The choice between these two classifiers would depend on the specific problem, the quality of the data, and the desired trade-off between robustness and computational efficiency.

Give one advantage and one disadvantage of encoding this as a one-hot vector instead of a numerical value?

Encoding categorical variables as one-hot vectors and numerical values are two common methods used to represent categorical data in machine learning.

One advantage of encoding categorical data as a one-hot vector is that it allows the model to learn a separate weight for each category, which makes it easier for the model to learn the relationship between the input variables and the output variables. This allows the model to handle categorical data with multiple levels, such as ordinal variables or categorical variables with many levels. Additionally, one-hot encoded categorical features are useful in the case of tree-based models as they are natively able to handle categorical variables.

One disadvantage of encoding categorical data as a one-hot vector is that it increases the dimensionality of the data, which can lead to the curse of dimensionality. This can cause problems such as overfitting, increased computational time and memory usage, and decreased interpretability of the model. Additionally, one-hot encoding can lead to sparse matrices, which can cause numerical instability in some algorithms.

On the other hand, encoding categorical variable as numerical values is easy to interpret, it does not increase the dimensionality of the data, which can be helpful in reducing the curse of dimensionality. However, it can be tricky when working with categorical variables that are ordinal, and the assumption that the difference between two category levels is equal may not always be true.

In summary, encoding categorical data as a one-hot vector allows the model to learn a separate weight for each category, making it easier for the model to learn the relationship between the input and output variables. However, it can lead to increased dimensionality and sparsity, which can cause problems such as overfitting, increased computational time and memory usage, and decreased interpretability of the model. On the other hand, encoding categorical variables as numerical values does not increase the dimensionality of the data and it is easy to interpret but it can be tricky when working with ordinal variables.

Let's say you decide to assign a numerical value between 0 and 1 to each answer from the

set but you hesitate between the 2 following options:

a. [0 (walk), 0.25 (bike), 0.5 (motorbike), 0.75 (pub. tr.), 1.0 (car)]

b. [0.9 (walk), 1.0 (bike), 0.7 (motorbike), 0.5 (pub. tr.), 0 (car)]

Option (a) assigns a random equidistant value to each possible answer, option (b) supposes that biking consumes more energy than walking or driving a car and assigns the values according to human perception.

Explain briefly: Is one option better than the other for a KNN ? and for a Tree-based algorithm*?

Encoding categorical variables as numerical values can be useful in certain machine learning algorithms, such as k-Nearest Neighbors (KNN) or tree-based algorithms. The choice between encoding options a or b depends on the specific problem and the type of algorithm being used.

Option (a) assigns a random equidistant value to each possible answer. This approach is useful when the categorical variable does not have any inherent order or when the difference between the categories is not known. It is useful in cases where the algorithm is not sensitive to the specific values assigned to each category and only cares about the relative distances between them, such as in KNN algorithm.

Option (b) supposes that biking consumes more energy than walking or driving a car and assigns the values according to human perception. This approach is useful when the categorical variable has an inherent order or when the difference between the categories is known. This approach can be useful in tree-based algorithms, such as decision trees or random forests, which are sensitive to the specific values assigned to each category.

In summary, for a KNN, option (a) is usually preferred as the algorithm is not sensitive to the specific values assigned to each category, only cares about the relative distances between them. On the other hand, for a tree-based algorithm, option (b) is usually preferred as the algorithm is sensitive to the specific values assigned to each category and can use this information to make more accurate predictions.

Tree-based algorithm

As I mentioned before, for a tree-based algorithm, option (b) is generally preferred, as it assigns numerical values to the categories based on an inherent order or known difference between them. This allows the algorithm to make more accurate splits and predictions by using the specific values assigned to each category.

For example, in a decision tree, when a categorical variable is split, the algorithm will compare the values assigned to each category to determine which category should be in which branch. If the values assigned to each category are arbitrary, such as in option (a), the algorithm may

what is tree- based algorithms

Tree-based algorithms are a class of machine learning algorithms that build a decision tree to make predictions. Decision trees are a type of model that represent a series of decisions, each represented by a node in the tree, that lead to a final prediction. Each internal node of the tree corresponds to a feature of the input data, each branch corresponds to a value of that feature, and each leaf node corresponds to a final prediction.

Tree-based algorithms are widely used in a variety of applications, such as classification and regression problems, because they are easy to interpret, can handle both categorical and numerical features, and are able to model complex non-linear relationships. The most common tree-based algorithms include:

Decision Trees: A basic tree-based algorithm that recursively splits the data into subsets based on the features and values that provide the most information gain.

Random Forest: An ensemble algorithm that builds a set of decision trees and combines their predictions to improve the overall performance of the model.

Gradient Boosting: An ensemble algorithm that builds a set of decision trees and combines their predictions in a stage-wise manner to improve the overall performance of the model.

XGBoost: An optimized version of gradient boosting algorithm that is more efficient and has more functionality.

All these algorithms work by recursively splitting the data into subsets based on the features and values that provide the most information gain, in order to identify the most important features and relationships in the data. These algorithms are widely used in industry, especially in the field of data science and machine learning.

Valentín Guerrero Cano

Given the different types of layers that are found in a deep convolutional neural network intended to classify images among, 10 classes, like Lecun's LeNet . List the layer types as they would be stacked in a network from input to output.

A deep convolutional neural network (CNN) intended to classify images among 10 classes, like Lecun's LeNet, would typically have the following layers stacked in a network from input to output:

Input Layer: This is the first layer of the network and it is responsible for accepting the input images. The input images are typically preprocessed before being fed into the network, such as resizing and normalizing.

Convolutional Layer: This is the main building block of a CNN. It applies a set of filters to the input image to extract features such as edges, textures, and patterns. These filters are typically small and move across the image, creating a feature map. Each convolutional layer is typically followed by a non-linear activation function, such as ReLU.

Pooling Layer: This layer is used to reduce the spatial dimensions of the feature maps and extract the most important features. Pooling layers are typically used to reduce the computational cost of the network.

Fully Connected Layer: This is a dense layer that connects all the neurons in the previous layers and is responsible for making the final predictions.

Output Layer: This is the final layer of the network, it receives the output of the fully connected layer and outputs the final predictions. The output layer typically has 10 neurons, one for each class, and applies a softmax activation function to produce the final probabilities for each class.

It is worth noting that, in some architectures, dropout layers are used to prevent overfitting, by randomly dropping out neurons during training. Also, batch normalization is used to normalize the outputs of the neurons to make the training converge faster.

In summary, a deep convolutional neural network intended to classify images among 10 classes, like Lecun's LeNet, would typically have the following layers stacked in a network from input to output: Input Layer, Convolutional Layer, Pooling Layer, Fully Connected Layer, and Output Layer.

What is a Gaussian Layer?

A Gaussian layer is a type of layer that can be used in a neural network to add Gaussian noise to the input data. The layer applies Gaussian noise to the input data by adding random values sampled from a Gaussian distribution to the input. The Gaussian distribution is characterized by its mean and standard deviation.

Gaussian noise can be used to improve the robustness of a neural network by making it more resistant to overfitting and small perturbations in the input data. The noise can also be used as a regularization technique, as it adds some randomness to the input data and makes the network less sensitive to specific input values.

The Gaussian layer can be implemented as a custom layer in a neural network, and it can be used in any position of the network. Some researchers use it in the input layer, to add some noise to the images before being processed by the network, while others use it in the intermediate layers to make the network more robust to small perturbations in the input data.

In summary, a Gaussian layer is a type of layer that can be used to add Gaussian noise to the input data in a neural network. It can be used to improve the robustness of a neural network and as a regularization technique, and it can be implemented as a custom layer in any position of the network.

In vector quantization what is the relationship between K-means and competitive learning? What do they have in common?

Vector quantization is a method used to reduce the dimensionality of a dataset by replacing each vector with a prototype vector that represents a group of similar vectors. K-means and competitive learning are two methods used in vector quantization.

K-means is a clustering algorithm that is used to group similar vectors together. It works by initializing K centroids, where K is the number of clusters, and then iteratively updating the position of the centroids to minimize the sum of the distances between the vectors and their closest centroid. The final result is a set of K centroids that represent the cluster prototypes.

Competitive learning is a method that is used to learn the prototypes in a vector quantization problem. It is based on the idea that a prototype vector should be as different as possible from the other prototypes. In competitive learning, each prototype vector competes with the

other prototypes to be the closest to the input vector. The prototype that wins the competition updates its position to become closer to the input vector.

Both K-means and competitive learning have in common that they are used to learn the prototype vectors in a vector quantization problem. They both aim to group similar vectors together and they both use a distance-based criterion to measure the similarity between vectors. However, the main difference between them is that K-means is based on a centroid-based approach while competitive learning is based on a prototype-based approach.

In summary, K-means and competitive learning are two methods used in vector quantization. Both methods are used to learn the prototype vectors in a vector quantization problem and they both aim to group similar vectors together. The main difference between them is that K-means is based on a centroid-based approach while competitive learning is based on a prototype-based approach.

How they will converge on a given initialization, and under which conditions?

K-means and competitive learning are both iterative algorithms that are used to learn the prototype vectors in a vector quantization problem. They both converge on a given initialization under certain conditions.

K-means algorithm is guaranteed to converge to a local optimum, under the condition that the initialization of the centroids is done in a random way and the cluster assignment step is deterministic. The algorithm iteratively updates the position of the centroids by computing the mean of the vectors assigned to each centroid and reassigning the vectors to the closest centroid. The algorithm stops when the centroids no longer change or when a maximum number of iterations is reached.

Competitive learning algorithm also converges to a local optimum under certain conditions. The algorithm is sensitive to the initialization of the prototype vectors and requires a large number of input vectors to converge. The algorithm iteratively updates the position of the prototype vectors by moving the winning prototype closer to the input vector. The algorithm stops when the prototype vectors no longer change or when a maximum number of iterations is reached.

It is worth noting that, both algorithms are sensitive to the initialization and the presence of outliers in the data, which can lead to poor results. Also, both algorithms are sensitive to the scale of the input data and it is recommended to normalize the data before applying them.

In summary, K-means and competitive learning are both iterative algorithms that are used to learn the prototype vectors in a vector quantization problem. K-means algorithm is guaranteed to converge to a local optimum under the condition that the initialization of the centroids is done in a random way and the cluster assignment step is deterministic. Competitive learning algorithm also converges to a local optimum under certain conditions, but it is sensitive to the initialization of the prototype vectors and requires a large number of input vectors to converge. Both algorithms are sensitive to the initialization and the presence

of outliers in the data, which can lead to poor results, also they are sensitive to the scale of the input data.

The result of K-fold Cross Validation is a numerical score. What does this score approximate?

The result of K-fold cross-validation is a numerical score that approximates the performance of a machine learning model on unseen data. The score is usually computed as the average of the model's performance on each fold, and it can be used to compare different models or to tune the hyperparameters of a single model.

The most common performance measure used in K-fold cross-validation is the accuracy, which is the proportion of correctly classified instances. Other performance measures can be used as well, such as precision, recall, F1-score, ROC AUC, etc. depending on the problem and the model.

The score obtained from K-fold cross-validation approximates the expected performance of the model on unseen data, also known as the generalization performance. By splitting the data into multiple folds and training and evaluating the model on different subsets of the data, we can get an estimate of how well the model will perform on unseen data. This is especially useful when dealing with small datasets, where we don't want to reserve a large portion of the data for testing.

In summary, the result of K-fold cross-validation is a numerical score that approximates the performance of a machine learning model on unseen data. It is computed as the average of the model's performance on each fold and it can be used to compare different models or to tune the hyperparameters of a single model. The most common performance measure used in K-fold cross-validation is the accuracy, but other performance measures can also be used depending on the problem and the model.

Explain why, for a similar complexity, the learned parameters of a RBFN might be more interpretable than the learned parameters of an MLP

A Radial Basis Function Network (RBFN) is a type of neural network that uses radial basis functions as the activation function for the hidden layer neurons. The RBFN has a clear and simple structure, with the hidden layer neurons being modeled as radial basis functions, which are typically Gaussian functions. Each neuron in the hidden layer is centered on a specific point in the input space and responds to the input vectors that are close to that point.

On the other hand, a Multi-layer perceptron (MLP) is a type of feedforward neural network that has one or more hidden layers between the input and output layers. The hidden layers consist of a large number of neurons, each with a non-linear activation function. The learned parameters of an MLP include the weights of the connections between the layers and the biases of the neurons in each layer.

Because the RBFN has a clear and simple structure with a small number of neurons and parameters, the learned parameters are more interpretable than the learned parameters of an MLP. The learned parameters of an RBFN can be directly visualized as the center of the

Gaussian functions, which are typically easy to interpret. In contrast, the learned parameters of an MLP are typically a large number of weights and biases that are not easily interpretable. Additionally, MLP's have the capacity to learn complex non-linear functions which makes it difficult to interpret the parameters.

In summary, for a similar complexity, the learned parameters of a RBFN might be more interpretable than the learned parameters of an MLP, because the RBFN has a clear and simple structure with a small number of neurons and parameters, which are easy to interpret, while the learned parameters of an MLP are typically a large number of weights and biases that are not easily interpretable.

For which of the following algorithms is data standardization mandatory (M), highly recommended (H) or optional (O): -K Nearest Neighbors - PCA - Linear Regression - Multi Layer Perceptron - SVM -RBFN

Standardization is a technique that is used to ensure that the data is on the same scale, so that features with large values don't dominate the results. The need for standardization depends on the algorithm being used, and it varies among algorithms.

K-Nearest Neighbors (KNN): Standardization is highly recommended (H) for KNN. KNN uses the distance between the instances to find the nearest neighbors, and if the data is not standardized, the distance will be dominated by the features with large values, leading to poor results.

PCA: Standardization is mandatory (M) for PCA. PCA is based on the correlation or the covariance matrix of the data, and if the data is not standardized, the results will be affected by the scale of the features.

Linear Regression: Standardization is optional (O) for Linear Regression. Linear regression is based on the minimization of the sum of squared errors, and if the data is not standardized, the results will be affected by the scale of the features. However, standardizing the data makes the interpretation of the coefficients easier.

Multi Layer Perceptron (MLP): Standardization is highly recommended (H) for MLP. MLP is a feedforward neural network, and if the data is not standardized, the learning process can be affected by the scale of the features, leading to poor results.

SVM: Standardization is highly recommended (H) for SVM. SVM uses the dot product between the instances to find the hyperplane, and if the data is not standardized, the dot product will be dominated by the features with large values, leading to poor results.

Radial Basis Function Network (RBFN): Standardization is optional (O) for RBFN. RBFN uses radial basis functions as the activation function, and if the data is not standardized, the results will be affected by the scale of the features. However, standardizing the data makes the interpretation of the centers of the radial basis functions easier.

In summary, standardization is mandatory for PCA, highly recommended for KNN, MLP, and SVM, and optional for Linear Regression and RBFN. Standardizing the data can make the

interpretation of the results easier, but it may not be necessary for all algorithms, depending on the problem

We want to classify handwritten characters (capital letters from A to Z) stored in square images with 22×22 pixels. The characters are not necessarily centered in the images. For the purpose of classification, the architecture of a deep convolutional neural network is designed. Each layer of neurons is here decomposed in operational layers (e.g., convolutional neurons consist of a convolution and an activation function). We have 5 layer in total, the first one apply convolution kernel 3×3 with stride 1, Max Pooling 2×2 with stride 2 and softmax activation function, the second and the third too. The Fourth I think apply fully connected scalar product (i am not sure) and softmax, and the fifth too. How many parameters are there in the complete network? Please, explain deeply all the operations you do.

A deep convolutional neural network (CNN) is a type of neural network that is commonly used for image classification tasks. The architecture you described includes 5 layers in total, with the first layer being a convolutional layer followed by a max pooling layer, and the remaining layers being fully connected layers.

The first layer applies a convolution operation using a 3×3 kernel and a stride of 1, which means that the kernel is moved one pixel at a time over the input image. The convolution operation creates a set of feature maps, where each feature map is the result of convolving the input image with a different set of weights (also called filters). The number of feature maps is determined by the number of filters used in the convolution operation.

After the convolution operation, a max pooling operation with a 2×2 kernel and a stride of 2 is applied. Max pooling is a down-sampling operation that reduces the spatial resolution of the feature maps by taking the maximum value of each pooling window. This operation helps to reduce the number of parameters in the network and to make the features more invariant to small translations of the input images.

The fourth and fifth layers are fully connected layers. A fully connected layer (also known as a dense layer) is a layer where every neuron is connected to every neuron in the previous layer. The output of each neuron in the previous layer is multiplied by a weight and then added to a bias term, to calculate the input for each neuron in the current layer. The fourth and fifth layers are both followed by a softmax activation function, which is a common choice for the output layer of

a classification network. It maps the output of the neurons to a probability distribution over the classes.

Now, to calculate the number of parameters in the complete network, we can consider the following:

- The first convolutional layer has a 3x3 kernel and a stride of 1. If we assume that we have n filters, each filter has $3 \times 3 \times 1 = 9$ parameters (3x3 for the kernel and 1 for the bias term). So, in total, the first convolutional layer has $9n$ parameters.
- The second and third layers are the same as the first, so they also have $9n$ parameters.
- The fourth and fifth layers are fully connected layers. If we assume that the fourth layer has m neurons and the fifth layer has o neurons, the fourth layer has $m \times (9n + 1)$ parameters ($9n$ for the weights and 1 for the bias term) and the fifth layer has $o \times (m + 1)$ parameters (m for the weights and 1 for the bias term).

We want to classify handwritten characters (capital letters from A to Z) stored in square images with 22x22 pixels. The characters are not necessarily centered in the images.

For the purpose of classification, the architecture of a deep convolutional neural network is designed. Each layer of neurons is here decomposed in operational layers (e.g., convolutional neurons consist of a convolution and an activation function).

Design the architecture yourself in the table below: Fill in all columns "L??" by ticking with a cross just one cell among "Fully" to "Smax" (see meaning below the table). Select the element with the best expected performance level.

Fill in the missing sizes in the row "size".

Fill in the missing number of neurons (row "#neu") for the output layer L5a-L5b.

Image L1a L1b L1c L2a L2b L2c L3a L3b L3c L4a L4b L5a L5b

Size 22 20 10 4 1 1 1 1 1 1

#neu 8 16 32 64

Fully

Conv

MaxP

AvgP

Sigm

Tanh

ReLU

Smax

Fully: Fully connected scalar product

Conv: Convolution 3*3 with stride 1

AvgP: Average Pooling 2*2 with stride 2

MaxP: Max Pooling 2*2 with stride 2

Sigm: Sigmoid

Tanh: Hyperbolic tangent

ReLU: Rectified linear unit

SMax: Softmax

The architecture you've described for a deep convolutional neural network for classifying handwritten characters is as follows:

- L1a: Convolutional layer with 3x3 kernel and stride 1, with 8 filters. The output feature maps have a size of 20x20.
- L1b: Max pooling layer with 2x2 kernel and stride 2. The output feature maps have a size of 10x10.
- L2a: Convolutional layer with 3x3 kernel and stride 1, with 16 filters. The output feature maps have a size of 8x8.
- L2b: Max pooling layer with 2x2 kernel and stride 2. The output feature maps have a size of 4x4.
- L3a: Convolutional layer with 3x3 kernel and stride 1, with 32 filters. The output feature maps have a size of 2x2.
- L3b: Max pooling layer with 2x2 kernel and stride 2. The output feature maps have a size of 1x1.
- L4a: Fully connected layer with 64 neurons.
- L4b: Softmax activation function
- L5a: Fully connected layer with 26 neurons (assuming there are 26 classes, one for each capital letter from A to Z)
- L5b: Softmax activation function

This architecture uses multiple convolutional layers with 3x3 kernel and stride 1 and max pooling layers with 2x2 kernel and stride 2, to extract the relevant features from the input images. The fully connected layers and the softmax activation functions are used to perform the final classification. The number of filters in each convolutional layer increases as the layers progress through the network, which can help to extract more complex features.

It is worth to mention that this architecture is a suggestion, and it may not be the best one for this specific task, other architectures such as ResNet, VGG, Inception, etc. could also perform well. The final performance depends on the quality and quantity of the data, the type of problem, the chosen optimizer and many other factors.