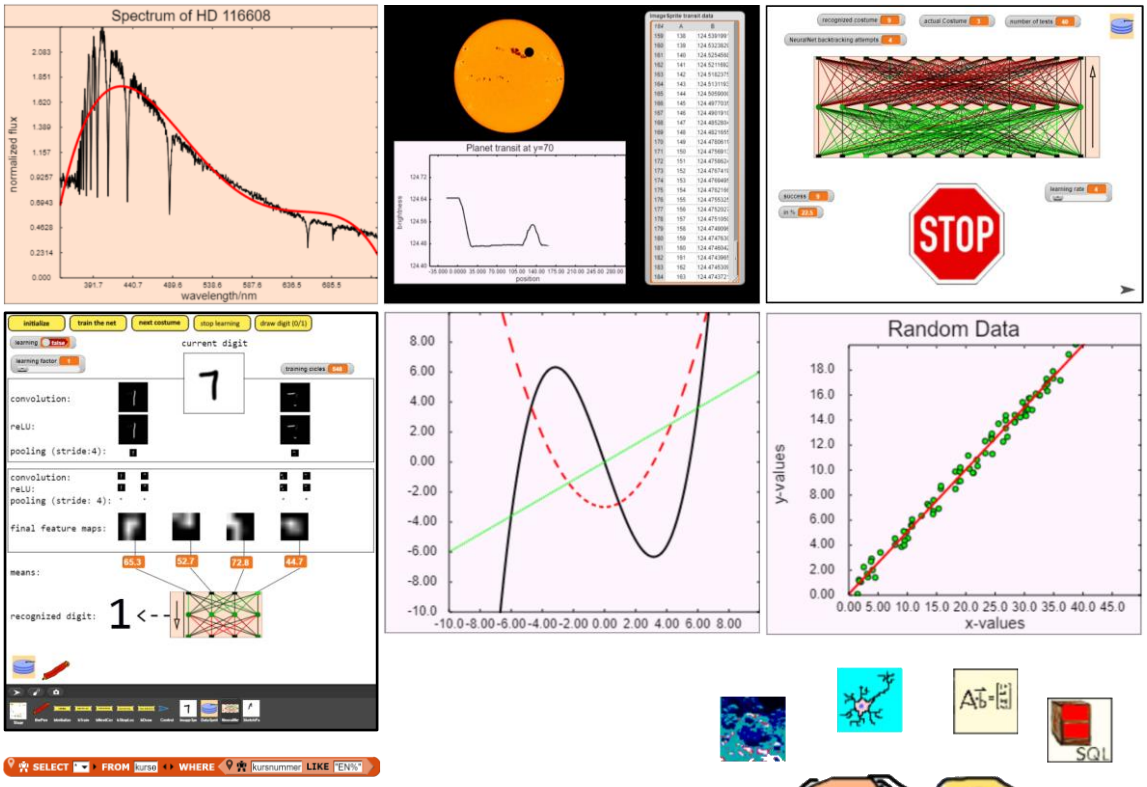
Eckart Modrow

Maschinelles Lernen mit den Snap! – ML.Sprites



emodrow@informatik.uni-goettingen.de

Die ML-Sprite-Prototypen und dieses Skript können geladen werden von

http://emu-online.de/MLSprites.zip bzw.
http://emu-online.de/MaschinellesLernenMitSnap.pdf

Prof. Dr. Modrow, Eckart:
Maschinelles Lernen mit *Snap!* – ML.Sprites

Wenn Sie mit diesem Buch zufrieden sind und ihre Anerkennung in Form einer Spende zeigen möchten, dann können Sie das auf folgendem PayPal-Konto tun:

emodrow@emu-online.de
Verwendungszweck: ML-Buch

Vorwort

Das vorliegende Skript beschreibt eine *ML.Sprite*-Bibliothek von *Snap!*-Blöcken, die für die (relativ) schnelle Verarbeitung großer Datenmengen gedacht ist. „Groß“ sind Datenmengen in Schule und Anfangsausbildung der Universitäten fast nie – weil sie noch vor einiger Zeit kaum frei zur Verfügung standen, und Geld ist im Ausbildungsbereich nun mal knapp. Inzwischen gibt es große Datenmengen aber zuhauf, sei es als Datensammlung im Netz oder eben als Bilddateien, denn die sind auch „groß“. Die Ausbildung hat damit die Chance, mit relevanten Daten umzugehen und damit zahlreiche Berührungspunkte zum Bereich „Informatik und Gesellschaft“ zu finden. Die sind auf lange Sicht unter Allgemeinbildungsaspekten ja wichtiger als irgendwelche Programmiertricks.

Gerade für Anfänger ist es wichtig, zu „sehen“, was sie mit ihren Programmierversuchen anrichten. Die fantastischen Visualisierungsmöglichkeiten von *Snap!* werden durch die *ML.Sprite*-Bibliothek um Blöcke für Grafiken und Bilder ergänzt, die ebenso wie die *Snap!*-Tabellen schnell das Ergebnis von Operationen anzeigen. Geschwindigkeit ist in diesem Bereich wichtig, weil sie experimentelles Arbeiten im trial-and error-Stil unterstützt. Muss man zu lange warten, dann probiert man nicht so viel aus. Die *ML.Sprite*-Bibliothek unterstützt dieses Vorgehen dadurch, dass die meisten zeitkritischen Funktionen in *JavaScript* implementiert sind. Nebenbei zeigen diese Blöcke auch, wie textbasiertes Programmieren sinnvoll in eine grafische Entwicklungsumgebung integriert werden kann.

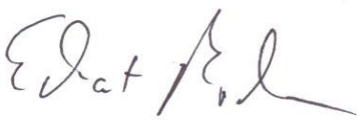
Die *ML.Sprite*-Bibliothek enthält Blöcke aus dem Bereich der Visualisierung von Daten und dem Umgang mit Tabellen, der durch die Einführung des Datentyps *table* unterstützt wird. Zusätzlich sind Funktionen der linearen Algebra mit den Datentypen *vector* und *matrix*, die Lösung linearer Gleichungssysteme und das Interpolieren durch Polynome vorhanden. SQL-Anfragen sind integriert, Neuronale Netze aus Perceptrons können einfach erzeugt und trainiert, Bildoperationen über Kernels sowie durch die Vektor- und Matrixoperationen schnell ausgeführt werden. Die Beispiele zeigen, wie das geschehen kann. Sie zeigen aber immer nur einen Weg – erfinden Sie für sich andere und bessere!

Die *ML.Sprite*-Bibliothek ist anders als die erste Version auf sechs Prototypen verteilt, die insgesamt als angefügte Teile des Gesamt-Sprites namens „*Arthur&Ina*“ oder auch einzeln geladen werden können. Dadurch wird die Anzahl der sichtbaren Blöcke begrenzt. Man kann sich auf die Prototypen beschränken, die gerade gebraucht werden. Jeder Prototyp enthält ein kleines Beispielskript, das seine Verwendung illustriert.

Ich bedanke mich sehr bei Jens Mönig und Rick Hessman für ihre Unterstützung und die zahlreichen Diskussionen und Anregungen.

Ansonsten wünsche ich viel Freude bei der Arbeit mit *Snap!* und den Prototypen der *ML.Sprite*-Bibliothek von Arthur und Ina!

Göttingen, am 13.1.2020



Inhalt

1 Künstliche Intelligenz und Schule

Der Begriff „Künstliche Intelligenz“ ist derzeit und auf absehbare Zeit mehr als aktuell. In Deutschland wurde das Wissenschaftsjahr 2019 zum „Jahr der künstlichen Intelligenz“ ausgerufen. Der Begriff prägt im Bereich der „Digitalisierung“ die Diskussion in Medien, Wirtschaft und Politik. Zunehmend finden sich zum Thema auch informatische fachdidaktische Beiträge.

In der Schulinformatik ist das Thema eigentlich nicht neu. Schon seit drei bis vier Jahrzehnten gibt es schulgeeignete Beispiele für z. B. zeichenerkennende neuronale Netze (NN), die von den Schülerinnen und Schülern selbst entwickelt und trainiert werden [Baumann] [Modrow1]. Solche Netze sind übersichtlich und gut verständlich, regen zu selbstständigem Arbeiten und danach zu auf fachlichen Erfahrungen beruhenden Diskussionen z. B. über die philosophischen Implikationen an [Modrow2]. Vor allem aber sind sie klein. Genau darin liegt der Unterschied zu den aktuellen NNs: die sind groß. Laut Ian Goodfellow [Deep Learning], einem der führenden Entwickler auf diesem Gebiet, hat sich grundsätzlich nichts gegenüber den alten kleinen Netzen geändert. Struktur und Methoden sind (fast) gleichgeblieben, aber natürlich verbessert worden. Geändert haben sich die Leistungsfähigkeit der Computer, auf denen die NNs laufen, und die Menge der vorhandenen Daten, mit denen sie trainiert werden können. Damit bleiben aber ältere Erkenntnisse gültig wie z. B. das aus 1967 von Marvin Minsky [Minsky] über die Äquivalenz von NNs und endlichen Automaten. Das ist auch kein Wunder, denn das Modell der endlichen Automaten hat seine Wurzeln eben in den ersten NNs. Solche Ergebnisse helfen aber bei der Einordnung eines Themas: denkt man beim Begriff „Lernen“ eher an Gehirne, bringt der endliche Automat das Gebiet eher auf die Ebene der Getränkeautomaten.

Wir haben damit aber ein Problem. In der Schule werden reale Anwendungen meist auf kleine Modellsysteme reduziert, die noch einige der ursprünglichen Eigenschaften zeigen. Wenn aber bei den aktuellen neuronalen Netzen die Eigenschaft, groß zu sein, gerade den Unterschied zu den früheren Versionen ausmacht, ist die Beschränkung auf kleine Netze zumindest fragwürdig. So etwas hätte – und hat – man ja schon vor einigen Jahrzehnten machen können. Was ist also neu an diesem Thema?

Die Vorschläge zur Behandlung großer NNs im Unterricht bestehen oft darin, dass fertige NNs mithilfe fertiger Trainingsdaten trainiert werden. Die Schülerinnen und Schüler können dann zusehen, wie das Netz lernt, also langsam seine Ergebnisse verbessert. Eigentlich benötigt man für diese Erfahrung kein reales NN, ein Video genügte auch. Man kann ja nicht sehen, dass das Netz groß ist, und aus dem Zusehen auch nicht erschließen, weshalb diese Größe von Bedeutung ist. Man sieht nur, dass sich die Ergebnisse verändern. Über NNs lernt man aus dieser Erfahrung allein nichts. Eine Diskussion der Auswirkungen der NNs beruht dann auf der Information, dass es sie gibt und dass sie lernen können. Weitere fachliche Grundlagen fehlen, sodass diese Diskussion ebenso gut in anderen Fächern stattfinden könnte.

Vergleichen wir die Situation einmal mit einem Beispiel aus der Physik. Das relativ neue Bild eines schwarzen Lochs [SZ] zeigt, dass es schwarze Löcher gibt und dass sie anscheinend Materie „verschlucken“. Diese Information allein integriert das Thema aber nicht in den Fachunterricht Physik, denn eine fachliche Behandlung schwarzen Löcher liegt weitgehend außerhalb der Möglichkeiten der Schule. Innerhalb eines Themenbereichs „Gravitation“, der zahlreiche Aktivitäten, historische und gesellschaftliche Bezüge, fachliche Probleme der Schulphysik usw. enthält, verknüpft das Bild aber die Schulphysik mit der „Wissenschaft nach der Schule“, zeigt Wege zu einer tiefgreifenderen Beschäftigung damit und regt z. B. zum Nachdenken darüber an, ob die Lernenden eine persönliche Perspektive auf diesem Gebiet sehen – oder eben nicht.

Was lernen wir daraus?

Das reine Vorstellen neuer Technologien hat in der Schule eigentlich nichts zu suchen – für Shows gibt es andere Kanäle. Die reine Information, dass es solche Technologien gibt, genügt ebenfalls nicht, das Thema einem bestimmten Fach zuzuordnen. Im Gegenteil: beschränkt man sich darauf, dann wäre es besser Fächern anzusiedeln, in denen z. B. die gesellschaftlichen oder philosophischen Auswirkungen diskutiert werden und das Thema so mit anderen Aspekten vernetzt wird. Erst die fachdidaktische Reduktion einer Fragestellung auf eine Komplexitätsebene, auf der in der Schule möglichst selbstständig und ideenreich von den Lernenden gearbeitet werden kann, macht das Thema pädagogisch fruchtbar.

> *Im Bereich der Künstlichen Intelligenz gehört damit nicht das passive Beobachten des Lernens der Netze in die Schule, sondern das aktive Fördern des Verständnisses der menschlichen Lernenden für die Grundlagen und Implikationen dieses Prozesses.*

Neu für die Schule sind die Werkzeuge, deren wir uns heute bedienen können. Die Visualisierungsmöglichkeiten einerseits und die Nutzung leistungsfähiger Bibliotheken andererseits machen es möglich, dass die Lernenden einfache erste Ansätze selbstständig erweitern und so Erfahrungen mit den Konsequenzen dieser Erweiterung machen können. Bei dieser Arbeit wird die Bedeutung der benutzten Begriffe deutlich und damit beurteilbar. Der Begriff des „Lernens“ hat z. B. auf dem Gebiet des Maschinellen Lernen weitgehend die Bedeutung „Parameteranpassung“. Das entspricht nicht so ganz der umgangssprachlichen Bedeutung – und damit seiner Verwendung z. B. in den Medien. Die Parameterzahl in z. B. kleinen Perzeptron-Netzen wächst mit ihrer Erweiterung immens – und damit die Zeit zu ihrem Training ebenso wie die Zahl der erforderlichen Trainingsdaten. Die Extrapolation zu wirklich großen Netzen wirft deshalb die Frage auf, wo diese Ressourcen herkommen. Bei der Trainingsdauer ist die Parallelisierbarkeit der Algorithmen und die Schnelligkeit der Rechner entscheidend, bei den Datenmengen aber ihre Quellen – und das sind oft wir. Die Anwendung von KI-Systemen kann uns also nicht egal sein, sie führt direkt zu Problemen mit dem Datenschutz und ist deshalb ein aktuelles gesellschaftspolitisches Thema. Das gilt ebenso für die Daten selbst. Die Arbeit zu Themen des maschinellen Lernens lehrt sehr schnell, dass mit den frei zur Verfügung stehenden Daten selbst gar nicht so viel anzufangen ist. Interessant wird es oft erst, wenn verschiedene Datenquellen verknüpft werden. Diese Verknüpfung erfolgt aber meist nicht über statistische Größen, sondern über die individuellen Quellen selbst – also uns. Hat z. B. die Ausbreitung einer Krankheit etwas mit dem Verhalten von Bevölkerungsgruppen zu tun? Das können wir eigentlich erst dann

schlüssig beantworten, wenn wir wissen, ob die Krankheit bei genau diesen Personen häufiger auftritt als bei anderen. Ansonsten kommen wir über Vermutungen nicht weit hinaus.

Die Beschäftigung mit maschinellem Lernen beansprucht Lernzeit – und die steht nur begrenzt zur Verfügung. Je mächtiger der benutzte Befehlssatz ist, desto mehr Zeit bleibt für die Frage nach den Konsequenzen. Die ML.Sprite-Bibliothek ist dafür gedacht, diese Zeit freizuräumen.

Dazu noch eine Anmerkung: Ich meine, dass neben dem üblichen Lernen von Fachinhalten und -methoden gerade im Bereich der Informatik ein gehöriger Teil Kreativität in den Unterricht gehört. Die Informatik stellt ja gerade dafür wunderbare Werkzeuge wie z. B. *Snap!* zur Verfügung. Konzentriert sich die Schule ausschließlich auf das Vermitteln von Fakten und Daten sowie das Einüben der Anwendung von Kalkülen, dann besteht die Gefahr, dass die Lernenden nie erfahren, wie es ist, Zusammenhänge und Hintergründe selbst zu entdecken und zu verstehen oder eigene Lösungen für interessante Problemstellungen zu finden und zu erproben. Das wäre etwas traurig, weil eine Chance zur Entwicklung einer kreativen, sich ihrer Möglichkeiten und Grenzen bewussten Persönlichkeit zumindest auf diesem Gebiet vertan würde. Das Ziel der *ML.Sprite*-Bibliothek ist es deshalb, den Lernenden einen Werkzeugkasten bereitzustellen, der für eigene Projekte im Bereich des maschinellen Lernens geeignet ist. Es ist ausdrücklich nicht das Ziel, fertige Lösungen für bestimmte Probleme zu liefern.

2 Maschinelles Lernen

Der Begriff „Maschinelles Lernen“ wird oft als Synonym für „Künstliche Intelligenz“ oder „Neuronale Netze“ benutzt. Diese Einschränkung stimmt aber nicht. Präziser ist z. B. die Definition, die sich auf der SAP-Seite [SAP] findet:

> *Die Technologie des maschinellen Lernens lehrt Computer die Ausführung von Aufgaben durch Lernen aus Daten, anstatt für die Aufgaben programmiert zu werden.*

Das „Lernen aus Daten“ kann dabei als Anpassung der Parameter einer Funktion verstanden werden. Einer Maschine wird ein Datensatz (Bild, Tabelle, Zeichenfolge, …) als Eingabevektor $\boldsymbol{E}$ präsentiert. Sie berechnet daraus einen Ausgabewert k, der die Eingabe einer Kategorie („Es handelt sich um eine Katze“, „Merkmal vorhanden“ (oder nicht), „das Wort ‚Auto‘“, …) zuordnet.

$$f(\boldsymbol{E}) = k$$

Diese Zuordnung kann auf sehr unterschiedliche Weise erfolgen. Man kann z. B. die Parameter eines Polynoms anpassen, ähnliche Eingabewerte suchen („k-nächste-Nachbarn“), mit Entscheidungsbäumen arbeiten, Bayes-Filter einsetzen, … – oder eben ein NN trainieren. Alle diese Verfahren haben die Gemeinsamkeit, dass die „Maschine“ einen Satz von Parametern enthält, die änderbar sind. Die Maschine „lernt aus Daten“, indem sie wiederholt einen Datensatz einliest, aus diesem mithilfe des aktuellen Parametersatzes den Ausgabewert berechnet und danach nach irgendeinem Verfahren diese Ausgabe mit dem „gewünschten“ Ausgabewert vergleicht. Gibt es da eine Abweichung, dann ändert sie die Parameter so, dass die Ausgabe dem gewünschten Wert zumindest näherkommt. „Gewünschte“ Werte können schon vorher bekannt sein („Das Bild ist ein Katzenbild“), von außen z. B. von einem „Trainer“ kommen („überwachtes Lernen“) oder von der Maschine selbst generiert werden („nicht überwachtes Lernen“), z. B. indem sie Merkmale aus vielen Trainingsdaten extrahiert („Clustering“). In allen Fällen „lernt“ die Maschine gar nichts, sondern sie passt Parameter nach einem vorgegebenen Verfahren an.

Auch dieses Vorgehen ist seit langem in den Schulen verbreitet. „Lernende“ Nimm-Spiele usw. finden sich schon in den ersten Informatik-Schulbüchern. Neu ist wiederum der Umfang der erforderlichen Trainingsdaten. Ein großes NN kann über Milliarden von Parametern verfügen, die trainiert werden müssen – und dafür sind dann auch „sehr viele“ Trainingsdaten vonnöten. Neu ist auch, dass diese Daten im Netz zur Verfügung stehen. Wenn also „umsonst“ zur Verfügung stehende Anwendungen „mit Daten“ bezahlt werden, dann wissen wir jetzt auch, wie und weshalb das geschieht.

Sichtet man gängige Lehrbücher zum maschinellen Lernen [Grus] [Albon], dann findet man dort folgerichtet gar nicht so viel über NNs, aber sehr viel über den Umgang mit Daten. Diese müssen z. B. normalisiert werden, um die vielen Eingabedaten, die aus sehr verschiedenen Quellen stammen können, kompatibel zu machen. Fotografieren wir z. B. viele Hunde mit einer älteren Digitalkamera und viele Katzen mit einer neueren, dann würde ein NN ohne Normalisierung der Daten aus diesen Bildern sehr wahrscheinlich lernen, dass Hundebilder kleiner als Katzenbilder sind.

Die Aufbereitung von Daten ist nun eine sehr handwerkliche Tätigkeit. Sie kann schrittweise erfolgen, erprobt und danach mit einfachen Algorithmen automatisiert werden. Die Erprobung wird stark erleichtert, wenn die Struktur der Daten leicht zu visualisieren, also z. B. in Tabellen oder als Graph darstellbar ist. Und Algorithmen sind einfach, wenn sie über eine übersichtliche Struktur verfügen, also z. B. nach einigen Vorbereitungsschritten aus einer Schleife bestehen, in der einige Alternativen mit den entsprechenden Anweisungen aufgezählt werden. Die Mächtigkeit der entwickelten Skripte hängt dann nicht so sehr von der algorithmischen Struktur wie von der Mächtigkeit der zur Verfügung stehenden Befehle ab. Oder umgekehrt: wenn man über genügend mächtige Befehle verfügt, dann kann man auch mit einfachen Programmen viel ausrichten. Die Anpassung der Parameter kann danach auf eine der gängigen Arten erfolgen. Stehen also entsprechende Werkzeuge zur Verfügung, dann ist die Aufbereitung von Daten ein sehr schulgeeignetes Thema. Die *ML.Sprite*-Prototypen sind als solche Werkzeuge gedacht.

Die Befehlssätze der ML.Sprites enthalten Lösungen für eine Reihe typischer Anfängerprobleme, z. B. das Sortieren, das Zeichnen eines Graphen oder die Darstellung eines Bildes in Falschfarben. Das bedeutet aber natürlich nicht, dass nichts mehr zu tun übrig ist – man kann nur komplexere Fragestellungen angehen. Statt also eine Liste zu sortieren kann man Lösungen für Probleme suchen, bei denen u. a. sortiert werden muss. Die Arbeit wird teilweise daraus bestehen, Befehlsfolgen aus Bibliotheksfunktionen zusammenzustellen, zu testen und anschließend als neuen Block zur Verfügung zu stellen, der z. B. von anderen Sprites zur Bewältigung anderer Aufgaben benutzt wird. Der Zugriff auf die Ressourcen anderer Objekte kann dabei als Übungsfeld für objektorientiertes Programmieren dienen – muss es aber nicht. Stattdessen kann auch mit einfachen Verfahren wie z. B. den Datenaustausch über globale Größen gearbeitet werden.

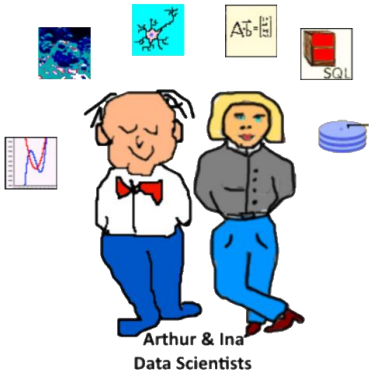
3 Der Aufbau der ML.Sprites

Der Aufbau der *ML.Sprites* orientiert sich an der Idee von dokumentierten Datensätzen, die aus zwei Teilen bestehen: den *Metadaten*, die die Struktur und den inhaltlichen Kontext der Daten beschreibt (z. B. Zahlenformat, Bilddimensionen, Aufnahmegerät, Aufnahmedatum, ...) und den dazugehörigen reinen *Datensegmenten*. Metadaten bestehen gewöhnlich aus *Dictionaries* - Namen mit zugewiesenen Werten (z. B. "Aufnahmedatum: 24.12.2018"). Beispiele für diese Struktur sind FITS-Dateien [FITS], die in der Astrophysik Standard sind, aber auch in der Vatikanischen Bibliothek Verwendung finden, oder JPEG Bilder vom Handy. Auch hier gibt es Metadaten (Bildgröße, Kompressionsgrad, Aufnahmedatum, oft auch GPS Koordinaten), ohne die eine Bilderzeugung nicht möglich wäre.

Wir adaptieren diese Struktur, indem wir einem *ML.Sprite* drei lokale Variable verpassen, die jeweils die Daten (*myData*), die Datenbeschreibung (*myProperties*) und ein Sammelbecken für (Fehler-)Meldungen aufgerufener Blöcke des Sprites (*myMessages*) enthalten. Diese Variablen können einerseits durch den Import von Daten aus unterschiedlichen Quellen (SQL-Abfrage, Textdatei, CVS-Datei, JSON-Datei, FITS-Datei, direkte Zuweisung, …) gefüllt werden, wobei die Eigenschaften *myProperties* den jeweiligen Daten anzupassen sind. Andererseits kann das auch „per Hand“ geschehen. Mithilfe dieser Eigenschaften können Daten in grafische Darstellungen (Graph, Datenplot, Histogramm, Bild, …) umgesetzt werden, wobei als Quelle entweder *myData* oder eine andere geeignete Tabelle gewählt wird.

Wichtig ist, dass die Bilderzeugung die Originaldaten nicht verändert. Wird also z. B. eine Aufnahme des Jupiters benutzt, um die Abstände seiner Monde zu bestimmen, dann müssen diese zumindest im Bild sichtbar sein. Dafür kann nach dem Einstellen einiger Parameter z. B. ein Falschfarbenbild erzeugt werden. In diesem wird der Jupiter selbst ziemlich unstrukturiert erscheinen. Will man dagegen das „Auge“ des Planeten genauer untersuchen, dann müssen die Parameter ganz anders gewählt werden, sodass die Monde wiederum kaum zu sehen sind. Alle diese Änderungen müssen in den Pixeln des aktuellen Kostüms des *Snap!*-Sprites geschehen, ohne die Bilddaten selbst zu beeinflussen.

Weil Tabellen in *Snap!* sehr schön dargestellt werden können, ist diese Darstellungsform nicht zusätzlich implementiert. Dafür ist der Datentyp *table* mit zahlreichen der im Bereich von *Data Science* üblichen Operationen (Tabellenoperationen, Korrelationsberechnung, affine Transformationen, Lösen linearer Gleichungssysteme, …) implementiert, der ausreichend schnell auch mit größeren Datenmengen umgehen kann.

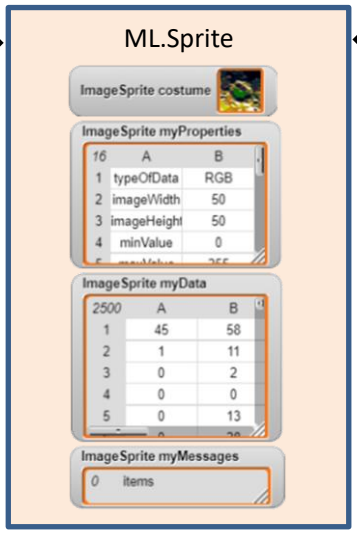


Da die Bibliothek (derzeit) 118 neue Blöcke enthält, wurden diese nach ihrer Funktionalität gruppiert und auf insgesamt sechs Sprites verteilt, die als Prototypen für unterschiedliche Aufgaben dienen können: ein *DataSprite* zum Umgang mit den eigentlichen Daten, ein *ImageSprite* zur Bildbearbeitung, ein *PlotSprite* für grafische Darstellungen, ein *NeuralNetSprite* für Perceptron-Netze, ein *SQLSprite* für Datenbankanfragen und ein *MathSprite* für Operationen der Linearen Algebra. Damit sie unabhängig voneinander einzusetzen sind, wurden einige Blöcke in leicht unterschiedlichen Varianten erstellt. Ein „Sammelbecken“ namens *Arthur&Ina* enthält diese Prototypen als Teile. Will man sie einzeln benutzen, dann können sie natürlich von *Arthur&Ina* gelöst und auch einzeln gespeichert werden.

Die Prototypen der Bibliothek haben die folgende Struktur:

Sie importieren Daten aus …

- Bild-Dateien
- Text-Dateien
- SQL-Abfragen
- JSON-Dateien
- CSV-Dateien
- …



Sie stellen Blöcke zur grafischen Darstellung der Daten, zur Bearbeitung von Tabellen, Lösen von Gleichungssystemen, Anwendung statistischer Operationen, … bereit.

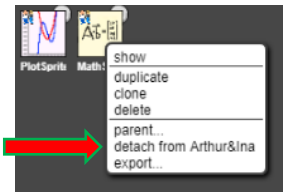
Die meisten Blöcke erhalten ihre Parameter (Bildgröße, Wertebereiche, Farben, …) aus dem Dictionary *myProperties*. Die mit ***set properties*** voreingestellten Eigenschaften ermöglichen es, ohne allzu viele Parameter Blöcke zum Erstellen von Grafiken, Diagrammen, … zu benutzen. Passen die Werte nicht, dann werden die Eigenschaften entweder einzeln (mit ***set property***) oder in Gruppen (z. B. mit ***set line attributes***) geändert.

Die Blöcke der einzelnen Sprites enthalten vorne jeweils ein anderes Symbol, um sie schnell voneinander und den Standardblöcken von *Snap!* unterscheiden zu können. Sobald man in *Snap!* eigene neue Paletten erzeugen kann, sollen die Bibliotheksblöcke dort untergebracht werden. Würde das schon jetzt in einer eigenen *Snap!*-Variante geschehen, dann würde man diese entweder laufend anpassen müssen oder sich von der *Snap!*-Entwicklung abkoppeln. Beides wäre unerfreulicher als die jetzt gewählte Lösung.

4 Arbeit mit den ML.Sprite-Prototypen

4.1 Versionen von ML.Sprites

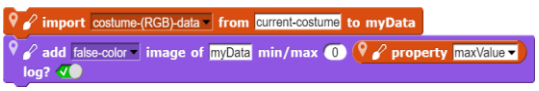
Arthur&Ina verfügen als erfahrene Data Scientists über das volle Programm der erforderlichen Methoden. Die entsprechenden Prototypen umschwirren ihre Köpfe, und klickt man doppelt darauf, dann landet man im entsprechenden Sprite.[1] Da wir beruflich anders orientiert sind als sie, werden wir kaum einmal alle Versionen der *ML.Sprites* gleichzeitig benötigen – aber welche wir brauchen, das hängt vom gewählten Problem ab. Wir können deshalb auch alle oder einzelne Sprites von *Arthur&Ina* lösen (Rechtsklick auf das Sprite und „*detach*“ auswählen) und diese einzeln speichern („*export*“ auswählen). Wenn wir Glück haben, dann hat das schon jemand gemacht und wir importieren nur noch die benötigten Prototypen (entweder mit „*import*…“ aus dem Datei-Menü oder durch „Reinziehen“ der Datei in das *Snap!*-Fenster). Das geht dann auch wesentlich schneller, als alles auf einmal zu laden. Danach sollten wir die Bühne im Werkzeug-Menü von *Snap!* mit „*Stage size*…“ auf eine geeignete Größe (je nach Bildschirmauflösung) einstellen, z. B. auf 800x600 Pixel.



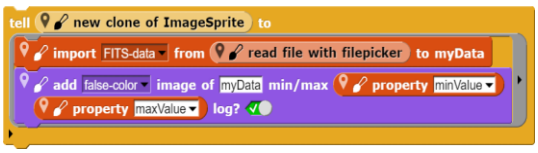
In den geladenen Sprites finden wir jeweils ein kleines Beispiel, das zeigt, wie sie genutzt werden können. Die neuen Blöcke verteilen sich auf die üblichen Paletten, meist auf *Looks*, *Operators* oder *Variables*. Sie stehen dort unten und tragen ein gemeinsames Symbol, um sie von den Standardblöcken zu unterscheiden.

Wenn wir wollen, dann können wir …

- … direkt im Prototypen arbeiten, die neuen Blöcke also ohne weitere Formalitäten einsetzen.
  Beispiel: Das aktuelle Kostüm, das z. B. von außen eingefügt wurde, wird in den Datenbereich *myData* eingelesen und die entsprechenden Eigenschaften werden gesetzt. Danach wird es in Falschfarbendarstellung in logarithmischer Darstellung ausgegeben, wobei der beim Import gerade ermittelte Maximalwert benutzt wird.



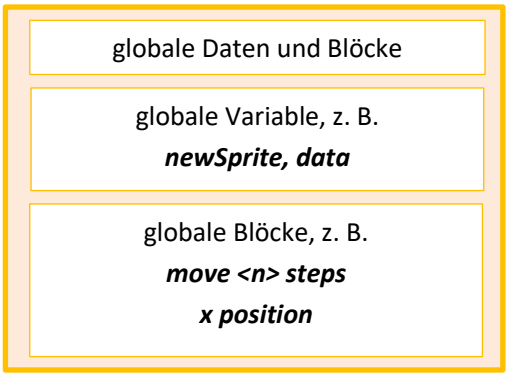
- … durch Rechtsklick auf das Sprite eine Kopie oder einen permanenten Klon des Prototyps erzeugen und genauso mit diesem arbeiten.
- … mit dem ***new clone of …***-Block aus der *Control*-Palette neue temporäre Klone erzeugen und diese im Weiteren benutzen.
  Beispiel: Zuerst wird ein temporärer Klon des *ImageSprites* erzeugt. Dieser wird gebeten, zuerst astronomische FITS-Daten einlesen zu lassen, wobei die zu lesende Datei vom Benutzer gewählt wird. Anschließend wird das Bild wie im anderen Beispiel angegeben angezeigt.

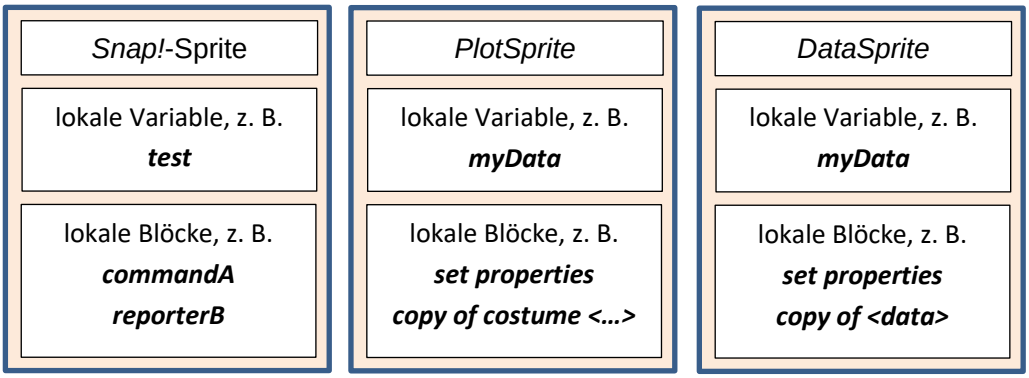


[1] Wenn man ehrlich ist, dann dienen *Arthur&Ina* nur dazu, alle sechs DataSprite-Prototypen gleichzeitig laden zu können.

4.2 Zugriff auf ML.Sprites

Findet man die gesuchte Funktionalität nicht im aktuellen *ML.Sprite*, dann stellt sich die Frage, wie „von außen“ auf die Daten und/oder Methoden eines anderen *ML.Sprites* zugegriffen werden kann. Da diese Größen alle lokal sind, kann man sie „von außen“ nicht direkt sehen. Das hat einerseits den Vorteil, die Anzahl der gerade sichtbaren Daten und Methoden übersichtlich zu halten, erschwert aber natürlich andererseits den Zugang. Wir gehen unterschiedliche Zugriffmöglichkeiten der Reihe nach durch. Dabei müssen wir deutlich zwischen lokalen und globalen Daten und Methoden unterscheiden. Als Beispiel wählen wir die Situation, dass ein „normales“ *Snap!*-Sprite die Möglichkeiten zweier Sprites von *Arthur&Ina*, eines *PlotSprites* und eines *DataSprites*, nutzen will.



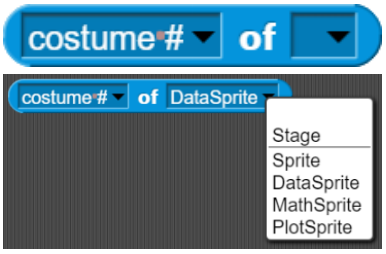


1. **Zugriff auf die Daten eines anderen Sprites**

Beispiel: Das *Snap!*-Sprite benötigt die Daten des *DataSprites*.

Lösung 1: Man weist im *DataSprite* einer globalen Variablen (***data***) den Wert der Daten (***myData***) zu. Auf diese kann man dann überall, auch im *Snap!*-Sprite, direkt zugreifen. (Sehr elegant ist das aber nicht!)

Lösung 2: Man greift direkt auf die Daten des *DataSprites* zu, indem man den ***of***-Block benutzt. Dazu wählt man zuerst im rechten Eingabefeld das gewünschte Sprite aus, hier: das DataSprite. Danach werden bei Klick auf das linke Eingabefeld neben ein paar Standardgrößen wie Position, Größe, … auch die lokalen Variablen (hier: myData, …) und die lokalen Methoden (hier: set property, …) aufgelistet (s. nächste Seite). Aus diesen kann man das Gewünschte auswählen.



In unserem Fall ist es die Variable ***myData***.

Mit dieser kann man dann wie üblich arbeiten, z. B. indem man das erste Element einer anderen Variablen zuweist.

Beispiel: Das *Snap!*-Sprite benötigt die Daten eines Klons eines *DataSprites*.

Lösung: Man bindet den Klon an eine lokale oder globale Variable. Über diese hat man dann auch später Zugriff auf den Klon.

Im Weiteren geht man vor wie im vorigen Beispiel, denn nur von „namentlich bekannten“ Sprites können die Eigenschaften aufgelistet werden. Erst zuletzt ersetzt man den Namen des ausgewählten Sprites (hier: DataSprite) durch die Variable, die auf den Klon zeigt.

Für Methoden wählen wir die folgende Konvention: globale Methoden sowie Methoden anderer Sprites ohne Parameter werden durch ***tell***- und ***ask***-Blöcke aufgerufen, Methoden anderer Sprites mit Parametern durch ***run***- und ***call***-Blöcke. (Aber manchmal machen wir es auch anders. 😉)

2. **Ausführung einer globalen Methode (command) durch ein anderes Sprite**

Beispiel: Das *Snap!*-Sprite sagt dem *PlotSprite*, dass es sich etwas weiterbewegen soll.

Lösung: Das *Snap!*-Sprite bittet das *PlotSprite* darum, sich zu bewegen, indem der oder die globalen Blöcke in den Skriptbereich eines ***tell***-Blocks gezogen werden. Auf der linken Seite des ***tell***-Blocks wird das adressierte Sprite (hier: das PlotSprite) ausgewählt. Die Blöcke werden dann im Kontext des anderen Sprites, also z. B. mit dessen aktueller Position und Richtung, ausgeführt.

Handelt es sich beim adressierten Sprite um einen Klon, dann geht man wie oben vor: man ersetzt zuletzt den Namen des *Plotsprites* durch die Variable.

3. **Ausführung einer globalen Methode (command) mit Parametern durch ein anderes Sprite**

Beispiel: Das *Snap!*-Sprite sagt dem *PlotSprite*, dass es sich unterschiedlich weit weiterbewegen soll.

Lösung: Der ***tell***-Block wird nach rechts um so viele Felder erweitert (kleiner Rechtspfeil), wie offene Parameter vorhanden sind. Die entsprechenden Eingabefelder bei den Methoden müssen vollständig leer sein!

Hinweis: Es geht auch anders und vor allem wesentlich differenzierter. Lesen Sie dazu das *Snap!*-Manual.

4. **Aufruf einer globalen Methode (reporter) durch ein anderes Sprite**

Beispiel: Das *Snap!*-Sprite erfragt die Properties des *PlotSprites*.

Lösung: 

Statt des ***tell***-Blocks wird der ***ask***-Block benutzt. Ansonsten wie oben geschildert.

5. **Aufruf einer lokalen Methode durch ein anderes Sprite**

Beispiel: Das Snap!-Sprite ändert Größe und Farbe des *PlotSprites*.

Lösung: Der ***run***-Block wird mit einer lokalen Methode benutzt, die mithilfe des ***of***-Blocks ausgewählt wird.

Beispiel: Ein *ImageSprite* wird gebeten, die Gesamthelligkeit um den angegebenen Punkt (100|50) im Radius von 5 Pixeln zu ermitteln.

Lösung:

Beispiel: Ein *MathSprite* wird gebeten, eine 4x3-Matrix mit Zufallszahlen bereitzustellen.

Lösung:

6. **Aufruf des Codes einer lokalen Methode durch ein anderes Sprite**

Beispiel: Das *PlotSprite* möchte die Methode des *ImageSprites* zum Zeichnen eines Kreises auf seinem eigenen Kostüm ausführen.

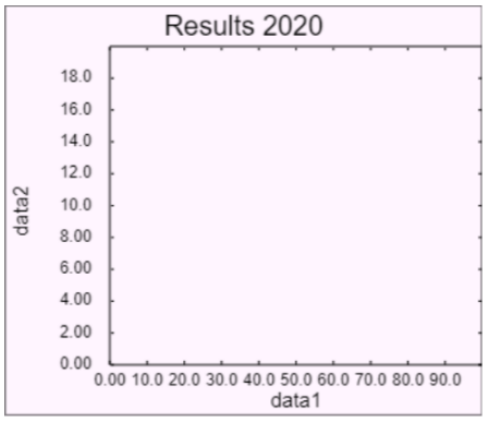
Lösung: Hängen lokale Methoden nicht von lokalen Variablen und/oder anderen lokalen Methoden ab, dann kann man den Code exportieren und in einem anderen Kontext ausführen. Typischerweise ist das bei *JavaScript*-Funktionen der Fall. Als Beispiele dienen hier die Zeichenoperationen des *ImageSprites*, die manchmal auch von anderen Prototypen benötigt werden. Da wir bei dem gewählten Verfahren, die Funktionalität an Sprites zu binden, keine globalen Methoden speichern können, sind die Zeichenmethoden des *ImageSprites* jeweils in einer zweiten Version vorhanden, die als „exportable“ gekennzeichnet wurde. Da hier nicht mehr auf die Properties des *ImageSprites* zurückgegriffen werden kann, „explodiert“ die Anzahl der erforderlichen Parameter. Aber dafür kann man den Code eben exportieren. 😉

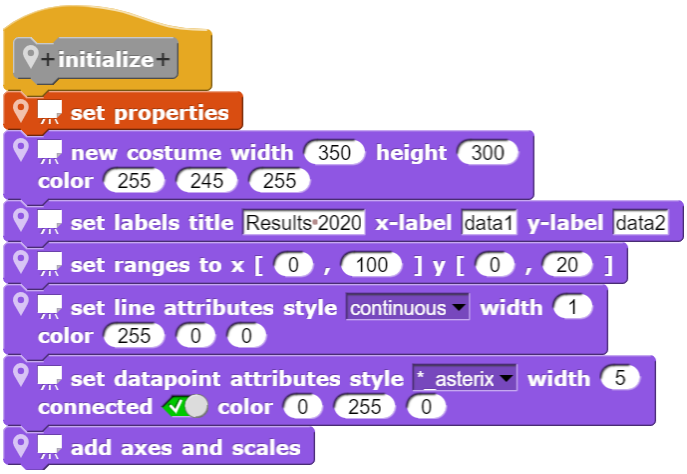
Im *PlotSprite* ist der Code des Zeichenfunktion über den ***of***-Block erreichbar.

Dieser Code wird im Kontext des *PlotSprites* ausgeführt, wobei die erforderlichen Parameter alle aufgelistet werden müssen. Anschließend wird das veränderte Kostüm wieder dargestellt.

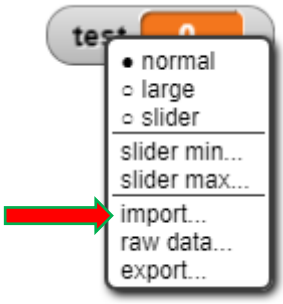
Muss man mehrere solcher Aufrufe kombinieren, dann wird es etwas umständlich. Es empfiehlt sich deshalb, innerhalb des adressierten Sprites eine lokale Methode zu schreiben, die die erforderlichen Aktionen auslöst. Diese wird dann von außen aufgerufen.

Beispiel: Die Vorgaben für ein Diagramm werden im *PlotSprite* eingestellt. In einem anderen Sprite kann man dann diese Methode einfach aufrufen.

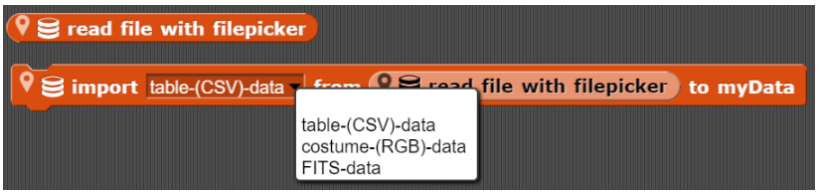




4.3 Importieren und Exportieren von Daten

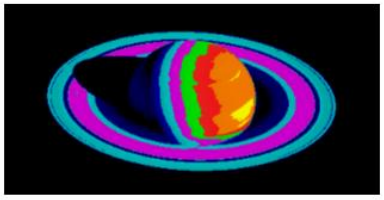


Snap! kann eine Reihe von Datenformaten direkt importieren. Das kann geschehen, indem man entsprechende Dateien auf das *Snap!*-Fenster „fallen“ lässt oder sie durch Rechtsklick auf einen Variablen-Watcher[2] importiert. Beides klappt gut mit Text-, CSV- und JSON-Dateien. Andere Text-Dateiformate wie FITS kann man ebenfalls so importieren, wobei nachgefragt wird, ob man es ernst meint. Das Exportieren funktioniert auf die gleiche Art. Will man dasselbe programmgesteuert machen, dann benutzt man den Reporter-Block ***read file with filepicker***. Es erscheint ein Dateimanager-Fenster, in dem man die Datei wie üblich auswählt. Danach werden die Daten importiert.



Als wesentliche Aufgabe bleibt anschließend, diese Daten der *myData*-Variablen zuzuweisen und die entsprechenden Eigenschaften in *myProperties* zu setzen. Das wird von dem folgenden Block erledigt, der Daten von außen in den *myData*-Bereich importiert. Dabei kann es sich um Bilddaten, Tabellendaten oder die Daten des aktuellen Kostüms handeln. Dieses wird als Tabelle von RGB-Werten gespeichert.

Beispiel: Durch das *ImageSprite* wird ein Bild (Quelle: [NASA]) gespeichert und mit Falschfarben neu dargestellt.



Beispiel: Knapp 600000 Datensätze aus einer CSV-Datei werden in etwa 10 Sekunden eingelesen. Die Eigenschaften werden gesetzt.

DataSprite myProperties

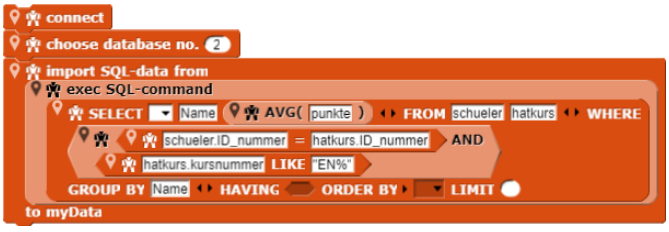
34	A	B
1	type	table
2	width	0
3	height	0
4	backColorRe	255
5	backColorGr	225

DataSprite myData

577704	A	B
1	tripduration	starttime
2	695	2013-06-01 (201
3	693	2013-06-01 (201
4	2059	2013-06-01 (201
5	123	2013-06-01 (201

Beispiel: SQL-Import

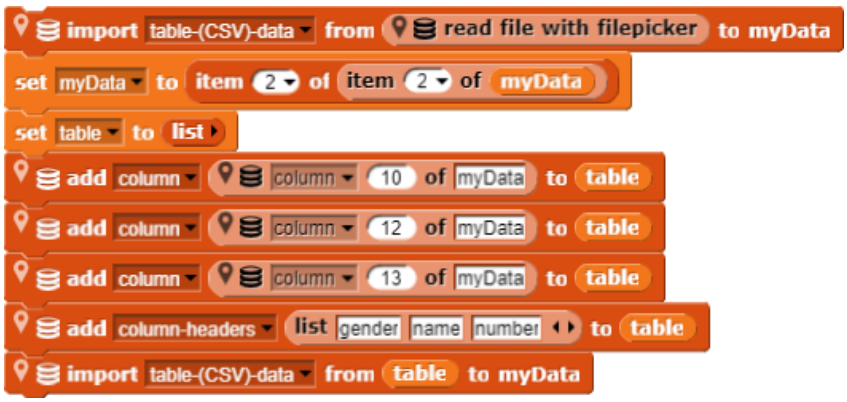
Haben wir Zugang zu einem SQL-Server, dann können wir auch von dort Daten einlesen. In unserem Fall importieren wir mithilfe des *SQLSprites* die Ergebnisse einer Abfrage in die Variable *myData*. Dabei werden die Daten in Tabellenform umgesetzt und ihre relevanten Eigenschaften wie Anzahl der Spalten und Zeilen, … neu gesetzt.

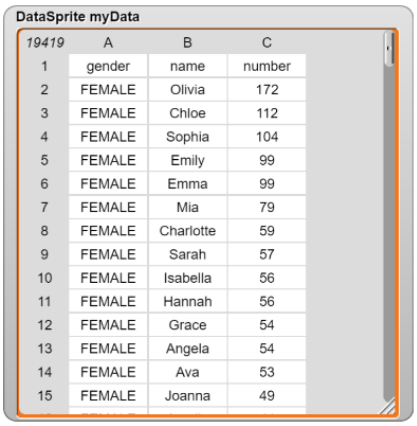


[2] Einen Variablen-Watcher erhält man, wenn man im Kästchen neben der Variablen einen Haken setzt.

Beispiel: JSON-Import

Der einfachste Weg ist auch hier, eine JSON-Datei einfach ins *Snap!*-Fenster „fallen“ zu lassen. Es geht aber auch automatisiert. Zuerst einmal suchen wir uns interessante JSON-Daten und wählen dafür natürlich die Statistik der Baby-Namen in New York City – was sonst. Der geeignete Block dafür ist wieder ***import <**table data**> from <**read file with filepicker**> to myData***. Das Ergebnis ist eine Liste mit zwei Spalten und zwei Reihen, den Metadaten und den eigentlichen Daten. Weil wir uns für die interessieren, ersetzen wir die Originaldaten durch das Element (2|2) der Tabelle. Natürlich haben wir uns vorher die einzelnen Elemente in Tabellenform angesehen, um zu prüfen, was wir da überhaupt geladen haben. Von den vielen Spalten kopieren wir die drei interessanten in eine neue Tabelle, fügen Spaltenüberschriften hinzu und importieren das Ergebnis wieder in *myData*.



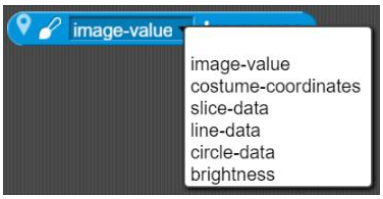


Das Ergebnis: 19419 Babynamen

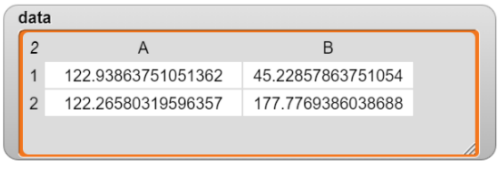
Wer hätte das gedacht!

Beispiel: Datenimport mit der Maus

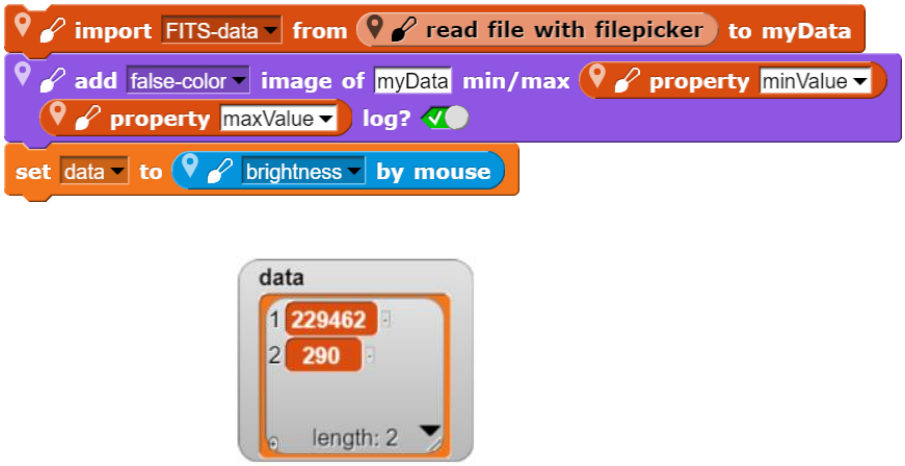
In vielen Fällen ist es gerade bei Bildern vorteilhaft, Daten mithilfe der Maus einzulesen. Dafür stehen in der *Sensing*-Palette des *ImageSprites* und teilweise auch beim *PlotSpite* Blöcke wie z. B. **<…> *by mouse*** zur Verfügung, mit denen Bildwerte, Bildkoordinaten, Koordinaten im benutzen Koordinatensystem für Graphen und/oder Datenpunkte, die Daten auf einem Schnitt durchs Bild, Anfangs- und Endpunkt einer Linie, Mittelpunkt und Radius eines Kreises und die summierten Helligkeitswerte zusammen mit deren Zahl in einem Kreis bestimmt werden können. Als Beispiel soll die Höhe antiker Säulen vermessen werden. Dazu wird das Kostümbild des *ImageSprites* mit den Säulen importiert und anschließend mit der Maus ausgemessen (gelbe Linie).

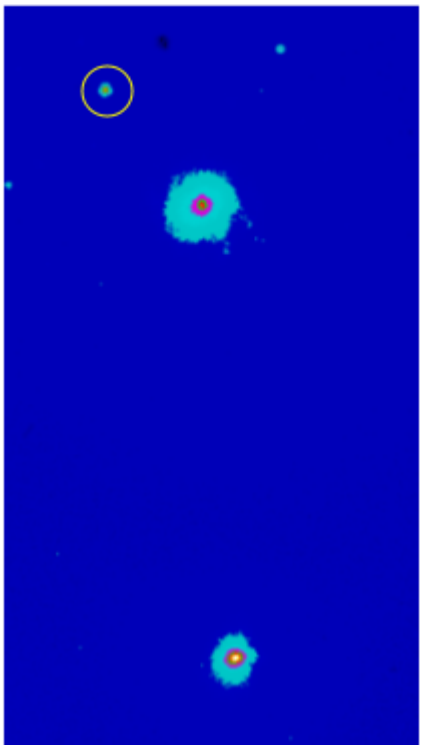


Beispiel: Messen von Abständen auf einem Bild



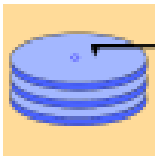
Beispiel: Messen der Gesamthelligkeit im Umkreis eines Pixels ein Bild (Quelle: [HOU])



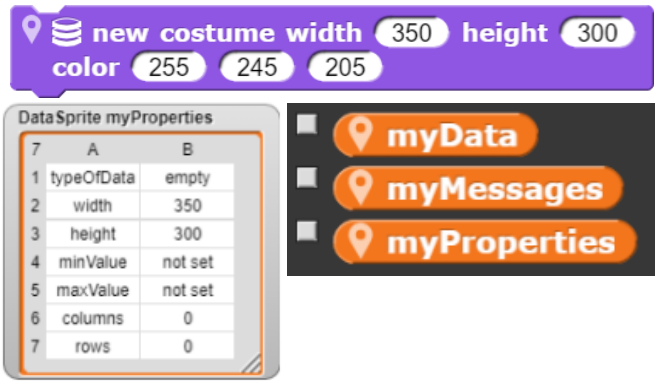


Der Export von Daten kann wiederum direkt aus einem Variablen-Watcher geschehen.

Für Skripte gibt es zwei neue Blöcke ***write <table> to CSV file <filename>*** sowie ***write string <string> to file <filename>***. Die Ergebnisse landen wie in *Snap!* üblich jeweils im Download-Ordner des Browsers. Die beiden Blöcke gestatten es, den Datenaustausch mit Tabellenkalkulationsprogrammen bzw. über Textdateien zu automatisieren, beispielsweise um Ergebnisse der Datenverarbeitung zu sichern.



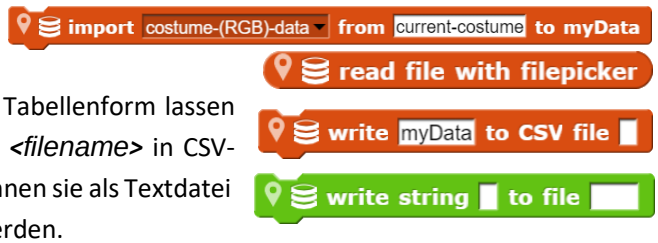
4.4 Das DataSprite



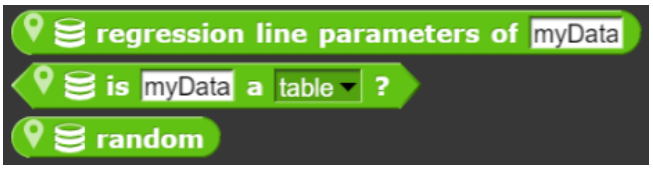
Alle *DataSprites* enthalten einen Block in der *Looks*-Palette, der ein Kostüm der angegebenen Größe mit den RGB-Werten der Hintergrundfarbe erzeugt und auf dieses umschaltet. Sie verfügen auch alle über den gleichen Satz von lokalen Variablen: ***myData*** enthält die eigentlichen Daten, ***myProperties*** deren Metadaten und ***myMessages*** eventuelle Mitteilungen, die beim Ausführen der Blöcke erzeugt werden – also meist Fehlermeldungen. Klappt etwas nicht, dann sollte man dort nachschauen.

	A	B
1	typeOfData	empty
2	width	350
3	height	300
4	minValue	not set
5	maxValue	not set
6	columns	0
7	rows	0

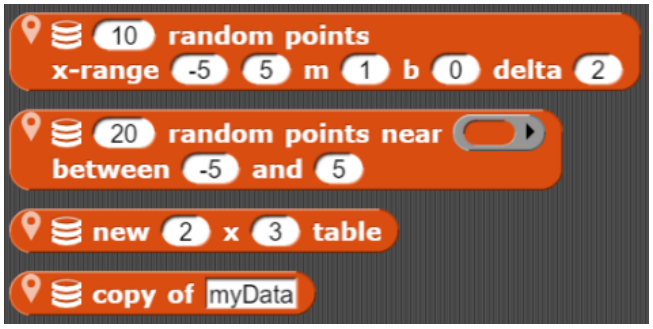
Das *DataSprite* dient zur Manipulation von Tabellendaten. Die Eigenschaften sind entsprechend angepasst: sie beschreiben neben dem Aussehen den aktuellen Zustand der gespeicherten Tabelle.



Zum Einlesen von Daten dient der Block ***import*** *<data>* ***from*** *<source>* ***to myData***, der oft zusammen mit ***read file with filepicker*** verwendet wird. Vorhandene Daten in Tabellenform lassen sich zur Weiterverarbeitung mit ***write*** *<table>* ***to CSV file*** *<filename>* in CSV-Dateien schreiben. Liegen sie als Zeichenkette vor, dann können sie als Textdatei mit ***write string*** *<string>* ***to file*** *<filename>* gespeichert werden.

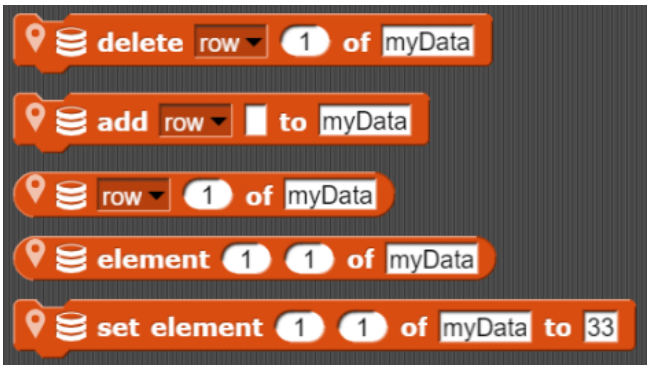


In der Operatoren-Palette finden wir drei weitere Blöcke: ***random*** liefert Zufallszahlen aus dem üblichen Bereich zwischen 0 und 1 mit voller Genauigkeit, ***regression line parameters of*** *<source>* berechnet die Parameter einer Regressionsgerade durch die angegebene Datenmenge. Das Prädikat ***is*** *<source>* ***a*** *<type>* testet die Eingabe darauf, ob es sich um eine *Tabelle*, einen *Vektor* oder eine *Matrix* handelt, wobei die letzteren nur Zahlenwerte enthalten dürfen. Der Block dient überwiegend zum Abfangen von Fehlern.

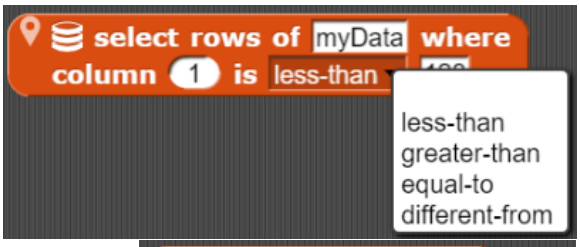


Die eigentlichen Funktionen des *DataSprites* finden sich in der Variables-Palette. Dort sind zwei Blöcke zur Erzeugung von Testdaten: *<n>* ***random points with ranges*** *<xmin><xmax>* ***and*** *<ymin><ymax>* erzeugt Zufallspunkte aus dem angegebenen Bereich, die um eine Gerade streuen, *<n>* ***random points near*** *<term>* ***between*** *<xmin>* ***and*** *<xmax>* entsprechend Punkte, die um den angegebenen Funktionsgraph verteilt sind. Der Block ***new*** *<n>* ***X*** *<m>* ***table*** erzeugt eine neue leere Tabelle der angegebenen Größe und ***copy of*** *<source>* liefert eine Kopie der übergebenen Daten. Die ist manchmal erforderlich, um zu verhindern, dass die Operationen mit den Daten die Originaldaten selbst verändern.

Die nächste Gruppe von Blöcken bezieht sich auf die Elemente der jeweils angegebenen Tabelle: ***delete*** *<row/column> <n>* ***of*** *<table>* löscht eine Spalte oder Zeile der angegebenen Tabelle, ***add*** *<row/column> <data>* ***to*** *<table>* fügt die übergebenen Daten als neue Zeile oder Spalte an die Tabelle an. Mit *<row/column> <n>* ***of*** *<source>* lassen sich einzelne Zeilen oder Spalten einer Tabelle kopieren. Die letzten beiden Blöcke gestatten den Zugriff auf einzelne Tabellenelemente.

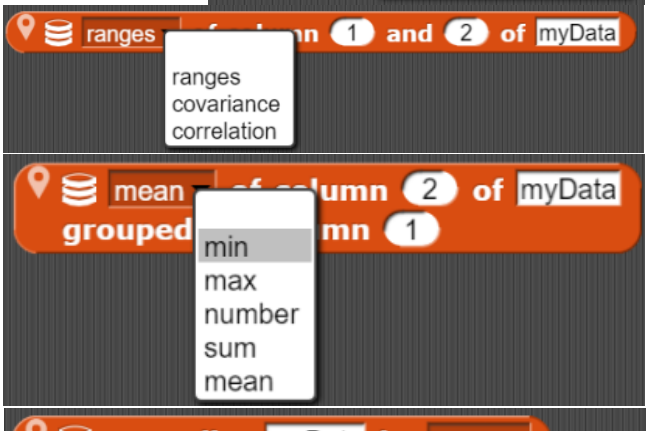


select rows of *<table>* ***where column*** *<n>* ***is*** *<predicate> <value>* erlaubt es, Tabellenzeilen mit bestimmten Eigenschaften auszuwählen.

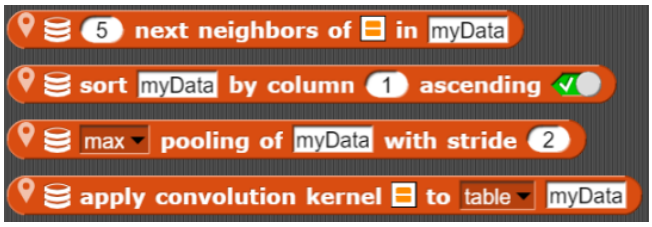


Der Block *<property>* ***of*** *<data>* ermittelt bei Bedarf Minimum oder Maximum eines Vektors sowie deren Positionen, die Anzahl der Elemente oder deren Summe, den Mittelwert oder Median sowie die Varianz oder die Standardabweichung.

Beziehungen zwischen zwei Spalten einer Tabelle lassen sich mit *<property>* ***of column*** *<n>* ***and*** *<m>* ***of*** *<table>* bestimmen. ***ranges*** ermittelt die Wertebereiche, die z. B. für grafische Darstellungen benötigt werden, die ***Kovarianz*** und der ***Korrelationskoeffizient*** werden für statistische Untersuchungen gebraucht. Der nächste Block gruppiert die Daten einer Spalte und berechnet dabei die angegebenen Größen der jeweiligen Gruppen.



Für grafische Darstellungen und den Vergleich von Daten ist der Block ***normalize*** *<data>* ***by*** *<option>* gedacht. Auch für Neuronale Netze wird er manchmal mit der Option ***softmax*** benötigt.



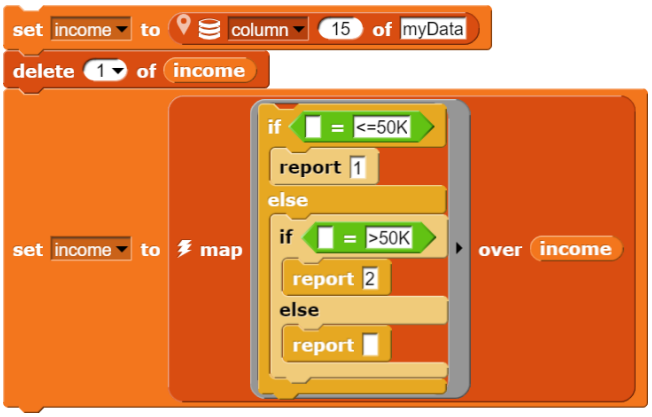
Die letzten vier Blöcke werden überwiegend für Beispiele aus dem Bereich des Maschinellen Lernens benötigt. Der Reporter-Block ***<k> next neighbors of <point> in <data>*** bestimmt die *k* nächsten Nachbarn eines Punktes in einer Punktmenge. Er kann z. B. bei *Clustering*-Problemen eingesetzt werden. ***sort <data> by column <n> <ascending/descending>*** gestattet das Sortieren einer Tabelle nach einer anzugebenden Spalte. Der ***pooling***-Block reduziert die Datenmenge mit anzugebender Schrittweite etwa bei *Convolutional Neural Networks* und die eigentliche *Faltung* kann für Bilder oder Neuronale Netze mit ***apply convolution kernel <kernel> to <table/image>*** erfolgen. Beispiele dafür folgen im nächsten Abschnitt.

Beispiel: Einkommensdaten aus dem US Census income dataset (Quelle: [Census])

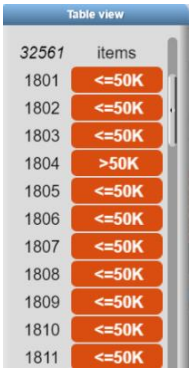
Wir wollen etwas in Daten wühlen und laden uns deshalb den *Census Income Dataset* aus dem Netz.[3] Die entsprechende CSV-Datei lässt sich aus dem Speicherverzeichnis in ***myData*** laden und sofort anzeigen. Sie umfasst 32562 Datensätze. Ein Rechtsklick darauf und die Wahl von *„open in dialog…“* zeigt alle Spalten.

Welche Zusammenhänge könnten sich nun darin zeigen?

Unsere *DataSprite*-Blöcke helfen erstmal nicht so sehr weiter, weil sie meist numerische Daten verarbeiten. Wollen wir sie einsetzen, dann müssen wir die Spalten so skalieren, dass sich numerische Inhalte ergeben. Im einfachsten Fall ersetzen wir Texte einfach durch Zahlenwerte – und sollten uns dabei gut überlegen, welche Folgen das bezüglich ihrer Interpretation haben könnte.



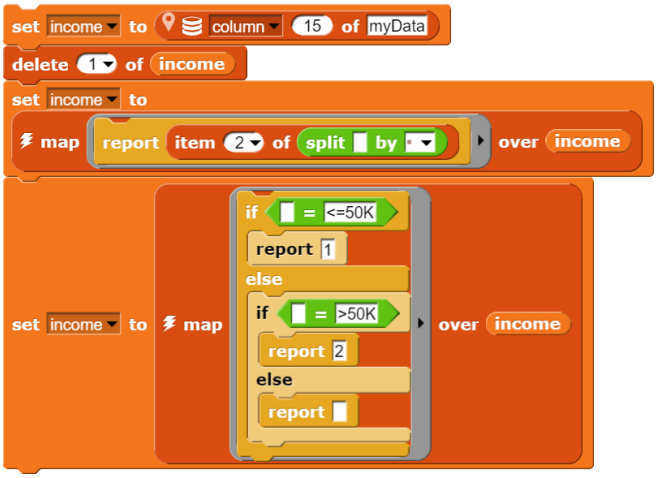
Fangen wir mit der letzten Spalte an: Die Einkommenswerte werden nur für zwei Bereiche angegeben: kleiner oder größer als 50000$. Wir ordnen diesen Bereichen die Werte 1 und 2 zu. (Oder 0 und 1, oder -1 und +1, oder 0 und 100, oder …. Hätten diese Änderungen Konsequenzen?) Um die Originalwerte nicht zu verändern, erzeugen wir eine Variable ***income*** und speichern dort die veränderten Werte, indem wir die Spalte 15 in diese Variable kopieren, den ersten Wert streichen (die Überschrift) und dann mithilfe des ***map…over…***-Blocks die Inhalte verändern. Jedenfalls versuchen wir das. Da es sich um recht viele Werte handelt, klicken wir rechts auf den ***map***-Block und wählen die just-in-time-Compilierung mit *„compile…“*. Es erscheint ein kleines Blitzsymbol vor ***map***. Leider erhalten wir nur die unveränderte Spalte 13, wenn wir uns das Ergebnis wieder als Tabelle ansehen.



Was ist los? Wir sehen uns das erste Element von ***income*** an und überprüfen, ob es sich um eine Zeichenkette handelt. Das ist der Fall, aber sie ist länger als gedacht:

[3] Dabei handelt es sich um einen der Trainingsdatensätze für Maschinelles Lernen.

Wir müssen also vorher die führenden Leerzeichen rauswerfen. Das klappt jetzt: unsere Variable ***income*** enthält nur noch die Werte 1 und 2, wie wir durch Ansehen schnell überprüfen können.



Wovon hängt dieses Einkommen nun ab?

Vielleicht vom Alter? Wir kombinieren die Spalte 1 (Alter) und unsere modifizierte Einkommensspalte zu einer neuen Tabelle namens ***testdata***.

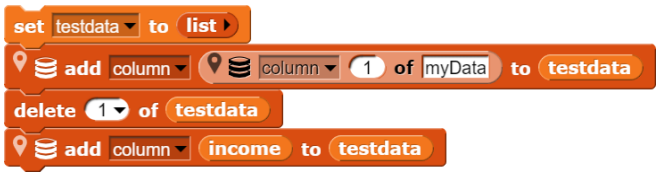


Table view

32561	A	B
12781	50	1
12782	68	1
12783	31	1
12784	22	1
12785	18	1
12786	25	1
12787	28	1
12788	17	1
12789	24	1
12790	20	1
12791	45	2
12792	44	1
12793	57	1
12794	33	2

Den Zusammenhang zwischen Alter und Einkommen beschreiben wir durch den Korrelationskoeffizienten. Die Berechnung ist einfach:

Und was bedeutet das?

Aufgaben:

1. Informieren Sie sich über die Bedeutung des Korrelationskoeffizienten und die Interpretation des erhaltenen Werts. Was bedeutet der Wert „0,2340…“?
2. Hängt der Korrelationskoeffizient in diesem Fall von der Art der numerischen Skalierung der Daten (1 und 2, -1 und 1, …) ab? Überprüfen Sie das.
3. Ermitteln Sie weitere Korrelationskoeffizienten, z. B. zwischen Ausbildung und Einkommen, Herkunftsland und Einkommen, Familienstand und Einkommen, Herkunftsland und Beruf, …
4. Informieren Sie sich darüber, ob und wann die Skalierung nichtnumerischer Daten einen Einfluss auf das Ergebnis haben kann.

Beispiel: New York Citibike Tripdata (Quelle: [NYcitibike])

Wir wollen mal nachsehen, wer in New York eigentlich Rad fährt. Dazu laden wir uns die Entleihdaten von NYCitibike eines Monats auf den Rechner, das sind die schon erwähnten knapp 600000 Datensätze. Die sehen wir uns genauer an.

Table view

577704	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	tripduration	starttime	stoptime	start station	start station	start station	start station	end station	end station	end station	end station	bikeid	usertype	birth year	gender
2	695	2013-06-01	2013-06-01	444	Broadway &	40.7423543	-73.9891507	434	9 Ave & W 1	40.7431744	-74.0036644	19678	Subscriber	1983	1
3	693	2013-06-01	2013-06-01	444	Broadway &	40.7423543	-73.9891507	434	9 Ave & W 1	40.7431744	-74.0036644	16649	Subscriber	1984	1
4	2059	2013-06-01	2013-06-01	406	Hicks St & M	40.6951284	-73.9959506	406	Hicks St & M	40.6951284	-73.9959506	19599	Customer	NULL	0
5	123	2013-06-01	2013-06-01	475	E 15 St & Irv	40.7352427	-73.9875856	262	Washington	40.6917823	-73.9737299	16352	Subscriber	1960	1
6	1521	2013-06-01	2013-06-01	2008	Little West S	40.7056925	-74.0167768	310	State St & Si	40.6892694	-73.9891286	15567	Subscriber	1983	1
7	2028	2013-06-01	2013-06-01	485	W 37 St & 5	40.7503800	-73.9833898	406	Hicks St & M	40.6951284	-73.9959506	18445	Customer	NULL	0
8	2057	2013-06-01	2013-06-01	285	Broadway &	40.7345456	-73.9907414	532	S 5 Pl & S 5	40.710451	-73.960876	15693	Subscriber	1991	1

Natürlich müssen wir uns bei der Quelle noch darüber informieren, was die Daten eigentlich genau bedeuten – also die Metadaten ansehen. Für das Geschlecht erfahren wir, dass 0: *unknown*, 1: *male* und 2: *female* bedeutet. Für die Spalten „*tripduration*“ und „*gender*“ ermitteln wir ein paar Daten:

Die mittlere Entleihdauer, bezogen auf das Geschlecht:

result

4	A	2
1	value	mean
2	0	1753.298818681775
3	1	1063.548722541860
4	2	1233.249445298994

Das dachten wir uns doch schon!

Für weitere Rechnungen löschen wir die Kopfzeile der Tabelle …

… und sehen mal nach, ob die am Broadway fauler sind:

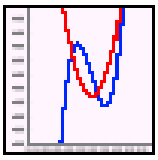
Aha. Wahrscheinlich ist es Central Park noch schlimmer!

Na gut. Alle Vorurteile müssen ja nicht stimmen. 😉

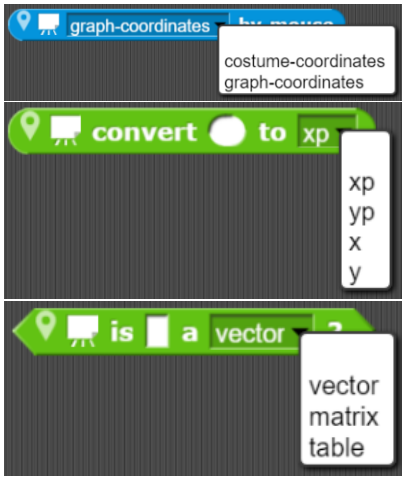
Aufgaben:

1. Vielleicht fahren aber nur die Frauen am Central Park mehr Rad. Überprüfen Sie das.
2. Am Central Park gibt es ja nicht nur eine Entleihstation. Ermitteln Sie geeignete Mittelwerte für den ganzen Bereich.
3. Gibt es eigentlich auch Entleihdaten für andere Stadtteile? Suchen Sie mal und vergleichen Sie die Ergebnisse mit Manhattan.
4. Ermitteln Sie die mittleren Entleihdauern pro Wochentag, insgesamt und für einzelne Stationen. Gibt es da Unterschiede? Weshalb?
5. Oben wurde die mittlere Entleihdauer bezogen auf das Geschlecht berechnet. Man könnte das auch umgekehrt machen. Wäre das völliger Unsinn oder gibt es Fragestellungen, bei denen das sinnvoll wäre?

4.5 Das PlotSprite

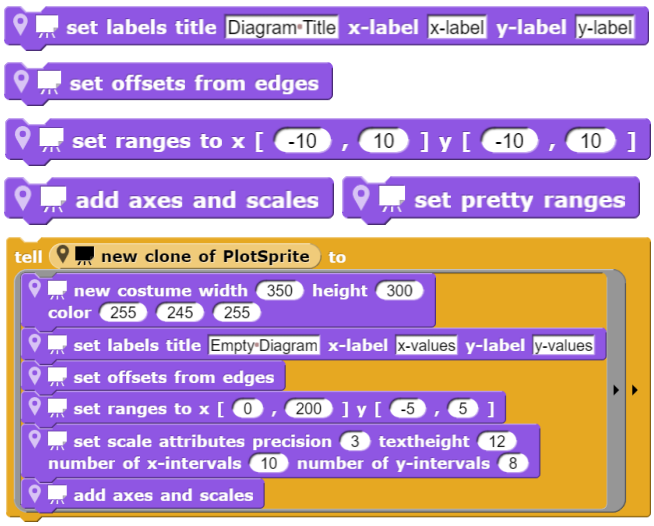


Das *PlotSprite* dient zur Erzeugung und Darstellung von Diagrammen. Man wird es deshalb meist zusammen mit einem *Data-Sprite* benutzen. Neben den drei Blöcken zur Verwaltung seiner ***Properties*** in der *Variables*-Palette, die in erster Linie Voreinstellungen zur Erzeugung der Diagramme enthalten, gibt es nur einen neuen Block ***<costume-/graph-coordinates> by mouse*** in der *Sensing*-Palette für das „Nachmessen“ von Werten mit der Maus und zwei neue Blöcke in der *Operators*-Palette ***convert <value> to <…>*** zur Umrechnung von Bildschirm- in Graph-Koordinaten und umgekehrt sowie den schon bekannten Prädikats-Block zur Typüberprüfung ***is <data> a <vector/matrix/table>?***.

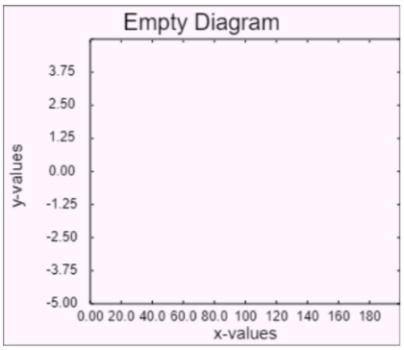


Die eigentlichen neuen Werkzeuge finden wir unten in der *Looks*-Palette. Einerseits kann man mit ihnen die voreigestellten Eigenschaften verändern und so an das aktuelle Problem anpassen, z. B. die Diagrammbeschriftungen oder die Wertebereiche. Andererseits kann man unterschiedliche Arten von Diagrammen erstellen.

set labels … setze die Werte für den Diagrammtitel und die Beschriftung der Achsen. Da von diesen die Abstände der Diagrammachsen von den Rändern abhängen, werden mithilfe von ***set offsets from edges*** auch diese Werte bestimmt. Wenn wir schon mal dabei sind, können wir auch gleich die Wertebereiche mit ***set ranges …*** und die Darstellung der Zahlen an den Achsen mit ***set scale attributes …*** neu einstellen. ***set pretty ranges*** setzt die Zahlenbereiche automatisch mit „hübschen“ Grenzwerten. ***add axes and scales*** zeichnet dann den Rahmen des neuen Diagramms.

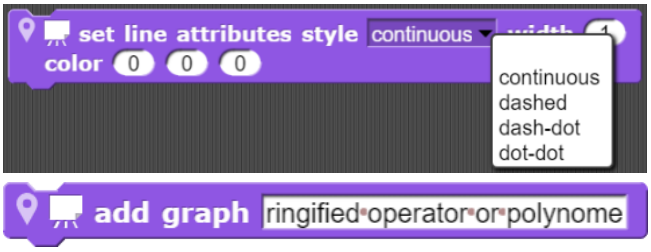


Da wir oft mehrere Diagramme benötigen, erzeugen wir zuerst einen Klon des *PlotSprites* und übergeben dem die erforderlichen Daten. Der stellt das geschmackvoll gestaltete neue Diagramm auf seinem Kostüm dar.

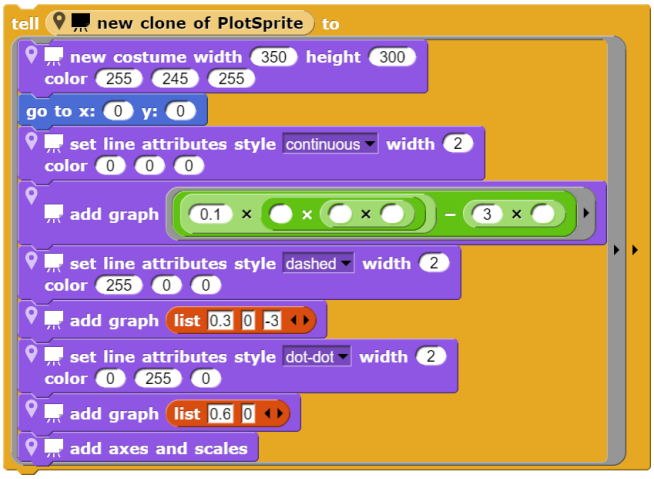


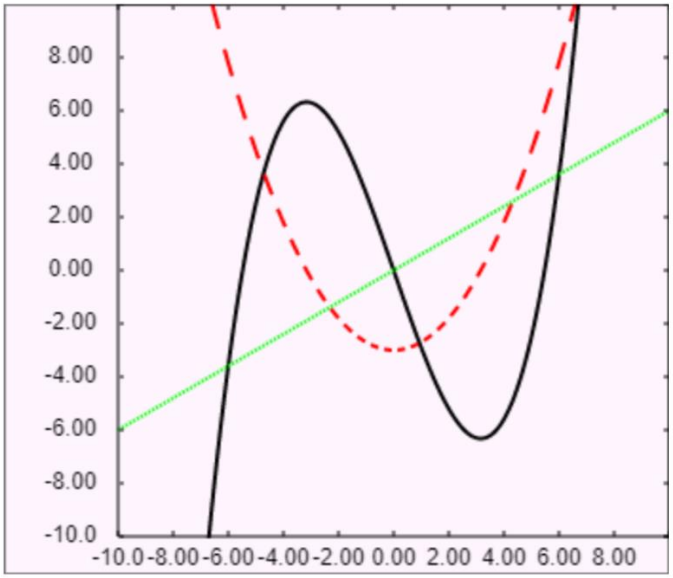
Drei der Blöcke in der *Looks*-Palette dienen eher dem Komfort bei der Programmierung: ***properties of group*** *<groupname>* fasst mehrere Eigenschaften zusammen und erleichtert so die Datenübergabe an *JavaScript*-Funktionen. ***ranges of*** *<data>* ermittelt die Wertebereiche einer zweidimensionalen Tabelle und ***copy of costume*** *<costume>* benötigt man, wenn man z. B. zwischen zwei Versionen eines Kostüms schnell umschalten will.

Die Diagramme selbst werden mit den restlichen neuen Blöcken erzeugt. Die Eigenschaften von Liniendiagrammen stellt man mit dem Block ***set line attributes …*** ein. Mit diesen Eigenschaften werden dann z. B Funktionsgraphen mit dem Block ***add graph …*** gezeichnet. Dabei kann der Funktionsterm entweder als Liste der Koeffizienten eines Polynoms oder als *„ringified“ Snap!*-Term übergeben werden.



Beispiel: Zeichnen einer Funktion und ihrer Ableitungen in unterschiedlichen Farben und Linienarten.





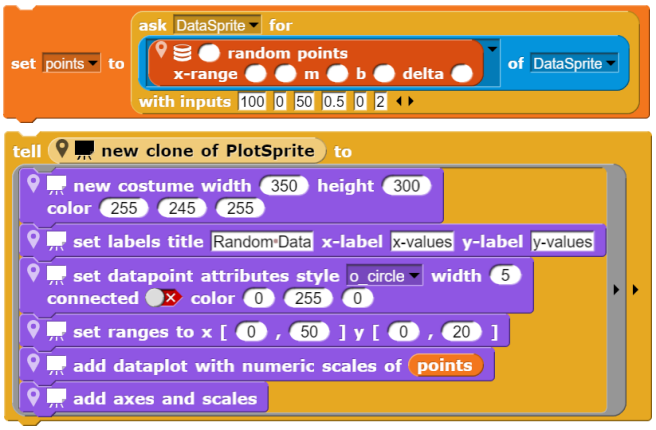
Aufgaben:

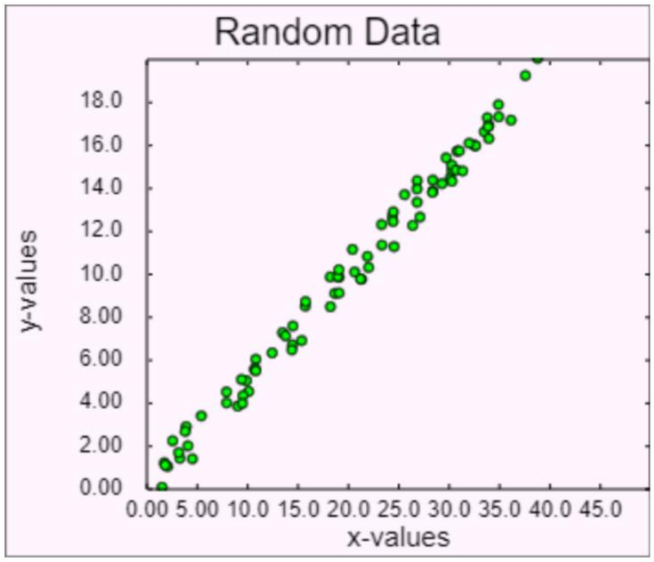
1. Stellen Sie unterschiedliche Funktionstypen (trigonometrische Funktionen, Logarithmen, Polynome, …) als Graph auf einem *PlotSprite* dar.
2. Ergänzen Sie die Funktionsgraphen durch deren Ableitungen.
3. Wählen Sie für die Darstellungen unterschiedliche Wertebereiche, Genauigkeiten der Zahlendarstellung und Textgrößen und Beschriftungen.

Wenn wir die Inhalte einer Datentabelle grafisch darstellen wollen, kann das mit dem Block ***add dataplot with numeric scales …*** geschehen. Skalen und Beschriftung werden wieder mit dem Block ***add axes and scales*** ergänzt. Die Art der Darstellung der Datenpunkte kann mit dem Block ***set datapoint attributes …*** sehr genau eingestellt werden. Die Linien dazwischen behalten die Linien-Attribute.

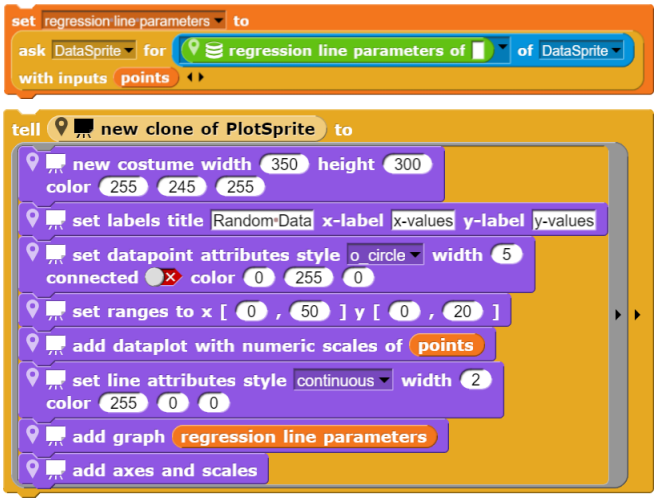
Beispiel: Darstellung einer Punktmenge

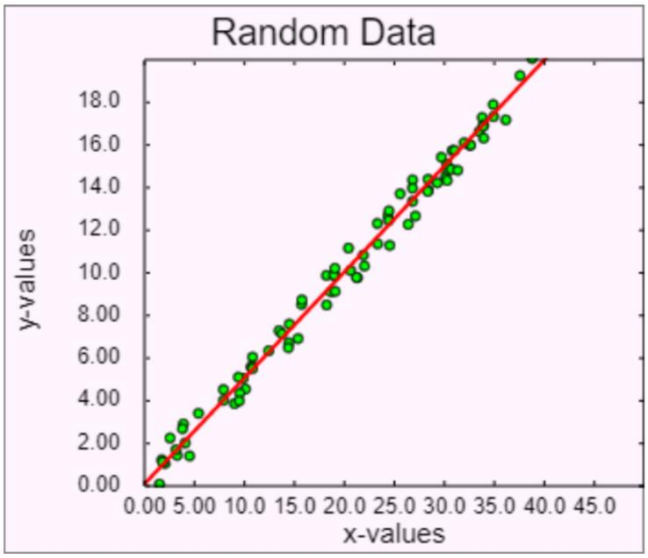
Wir bitten ein *DataSprite* um die Bereitstellung von 100 Zufallspunkten, die um eine Gerade mit der Steigung m=0.5 und dem Achsenabschnitt b=0 streuen. Die erhaltenen Punkte stellen wir in einem Diagramm dar.





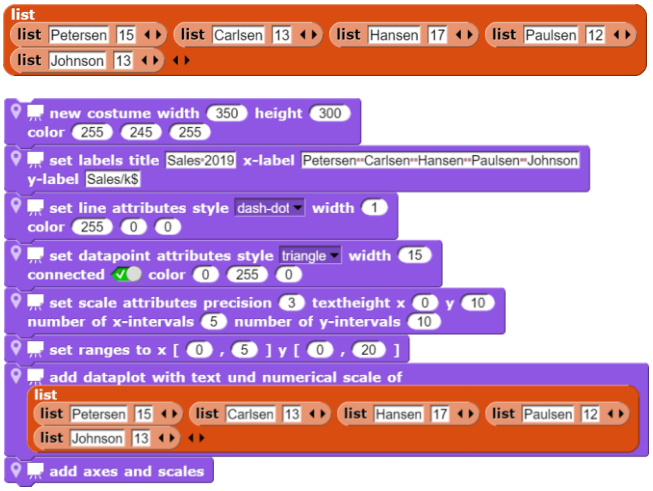
Beispiel: Zusätzlich wird jetzt noch die Regressionsgerade eingezeichnet.

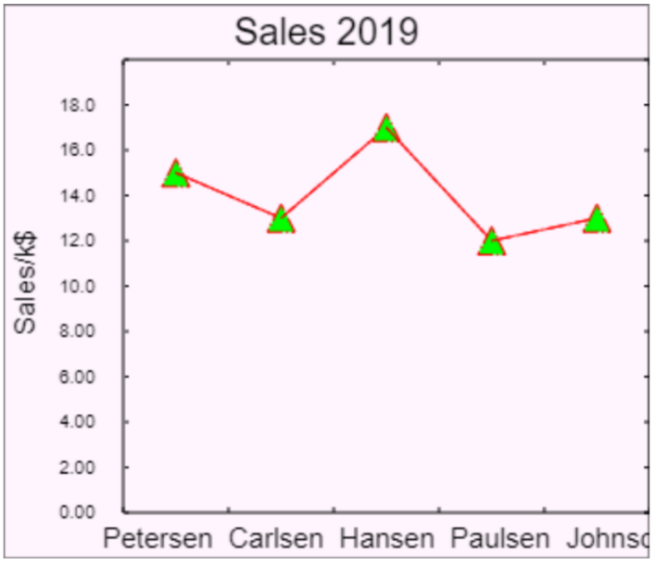




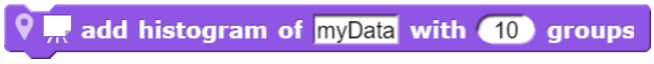
Beispiel: Darstellung gemischter Daten

Oft werden Textdaten mit numerischen Daten zusammengefasst. Ein Beispiel wären die Umsatzdaten verschiedener Vertreter in einem Jahr in einem Bereich. Wollen wir die grafisch darstellen, dann muss z. B. die x-Achse mit Textdaten beschriftet werden, während die y-Achse wie gehabt behandelt wird. Zum Erzeugen des Diagramms benutzen wir den Block ***add dataplot with text and numerical scale***.





Als letzten der neuen Blöcke des *PlotSprites* betrachten wir ***add histogram of*** *<data>* ***with*** *<n>* ***groups***. Histogramme lassen sich direkt aus Datenquellen erzeugen und darstellen.



Beispiel: Ein RGB-Bild wird geladen, in Graustufen zerlegt, und die normalisierte Verteilung der Bildwerte wird als Histogramm auf einem neuen *PlotSprite* dargestellt. Das eigentliche Bild finden wir als Kostüm eines zusätzlichen Sprites namens „*thePicture*“.

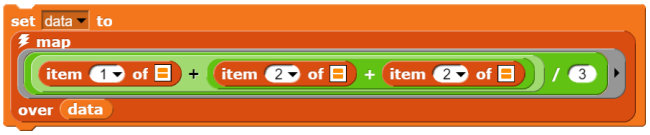
Zuerst einmal laden wir das Bild in den Datenbereich eines *DataSprites*:

Wir erhalten 172800 RBG-Werte.

DataSprite myData

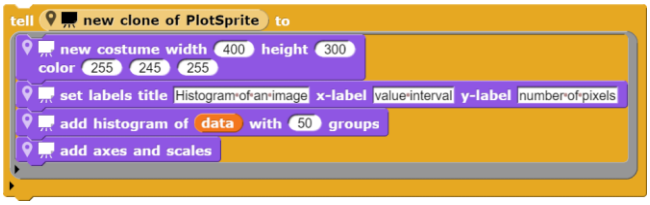
172800	A	B	C	D
1	152	182	217	255
2	148	178	209	255
3	154	178	208	255
4	152	178	211	255
5	151	177	212	255
6	153	177	215	255
7	155	180	213	255
8	154	179	212	255
9	159	183	210	255

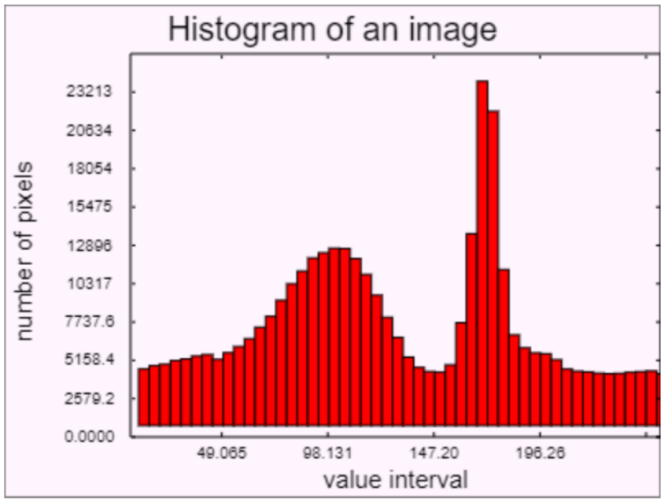
Diese wandeln wir in Grauwerte um. Dafür benutzen wir die compilierte Version des ***map***-Blocks.



Danach wechseln wir zum *PlotSprite* und kopieren die geladenen Daten des *DataSprites*.

Jetzt können wir die Daten als Histogramm z. B. auf einem neuen *PlotSprite* darstellen.





Aufgaben:

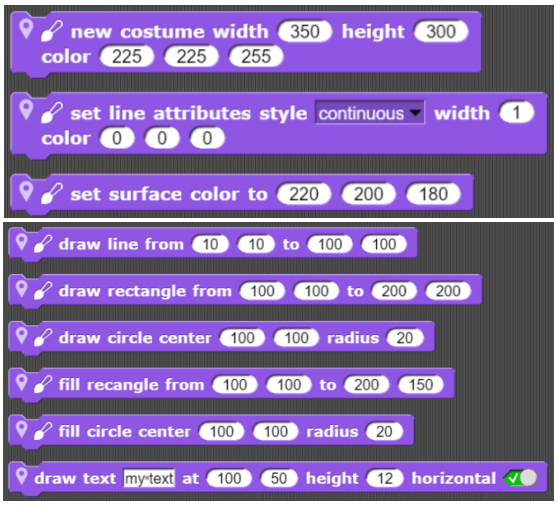
1. Suchen Sie im Netz unterschiedliche Datenmengen. Stellen Sie diese oder Teile von diesen grafisch dar.
2. Automatisieren Sie die Histogrammerstellung durch einen neuen Block ***histogram of <costume>***. Vergleichen Sie die Histogramme typischer Bildtypen. In wieweit ist ein Vergleich von Bildern auf diese Art möglich bzw. wo könnten Schwierigkeiten auftreten?
3. Stellen Sie im gleichen Diagramm die drei Farben eines RGB-Bildes durch Graphen und/oder Histogramme dar.

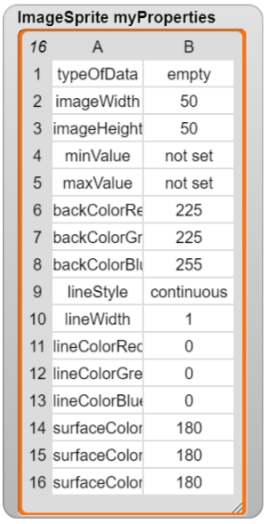
4.6 Das ImageSprite

Das *ImageSprite* dient zur Darstellung und Manipulation von Bildern. Dafür enthält es in der *Variables*-Palette neben den obligatorischen Blöcken zur Verwaltung der Eigenschaften die Möglichkeit, Bilder zu laden und die Minimal- und Maximalwerte der Bilddaten zu bestimmen.

In der *Looks*-Palette finden wir den schon bekannten Block ***add image …***, mit dem Daten auf dem Kostüm als Grau- oder Falschfarbenbild dargestellt werden können. Mithilfe des Blocks ***copy of costume*** lassen sich bei Bedarf Kostüme duplizieren.

Neben dem üblichen Block zur Erzeugung eines neuen Kostüms, über den alle *ML.Sprites* verfügen, gibt es zwei Blöcke zur Einstellung von Linieneigenschaften und Füllfarbe. Mit diesen und sechs Blöcken zum Zeichnen von Figur-Umrissen und gefüllten Figuren auf dem Kostüm lassen sich ganz gut z. B. Zufallsgrafiken erzeugen.

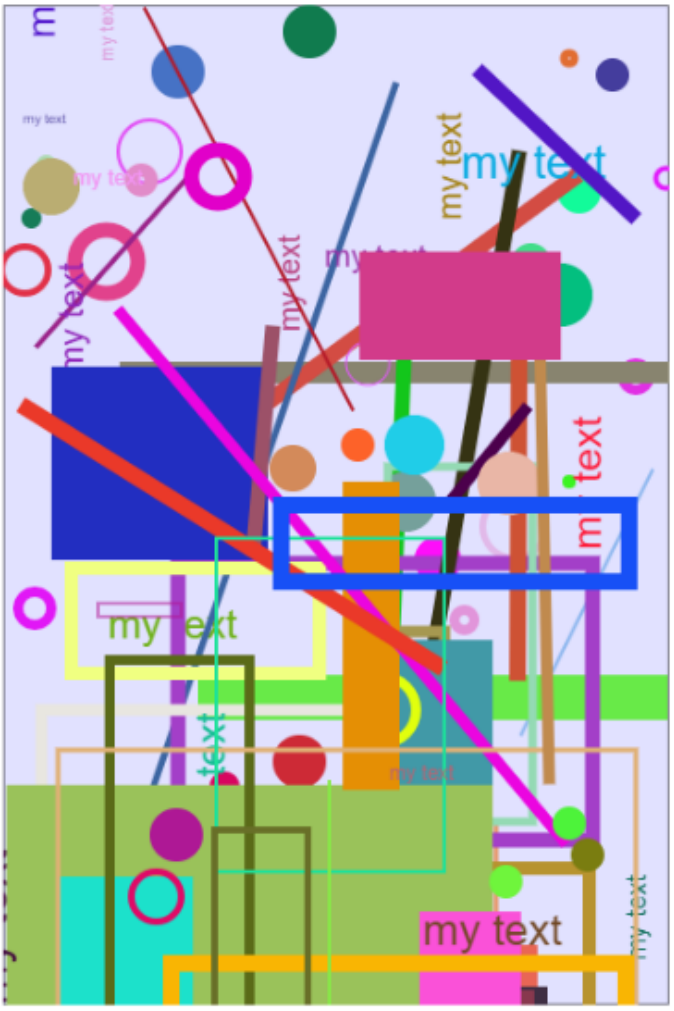




ImageSprite myProperties

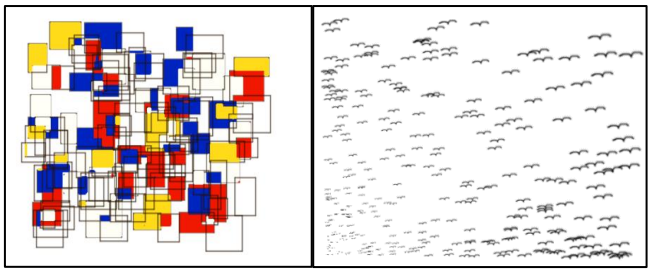
16	A	B
1	typeOfData	empty
2	imageWidth	50
3	imageHeight	50
4	minValue	not set
5	maxValue	not set
6	backColorRe	225
7	backColorGr	225
8	backColorBl	255
9	lineStyle	continuous
10	lineWidth	1
11	lineColorRec	0
12	lineColorGre	0
13	lineColorBlu	0
14	surfaceColor	180
15	surfaceColor	180
16	surfaceColor	180

Beispiel: Erzeugung von Zufallsgrafiken

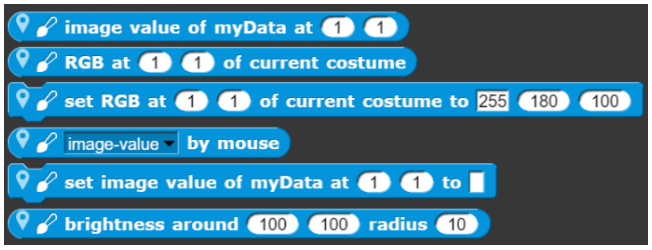


Aufgaben:

1. Suchen Sie im Netz nach Bildern von Piet Mondrian. Versuchen Sie, ähnliche Zufallsbilder auf dem *ImageSprite* zu erzeugen.
2. Mithilfe eines „Fluchtpunktes“ lassen sich Bilder erzeugen, in denen sich Objekte scheinbar „von hinten nach vorne“ bewegen. Versuchen Sie es.

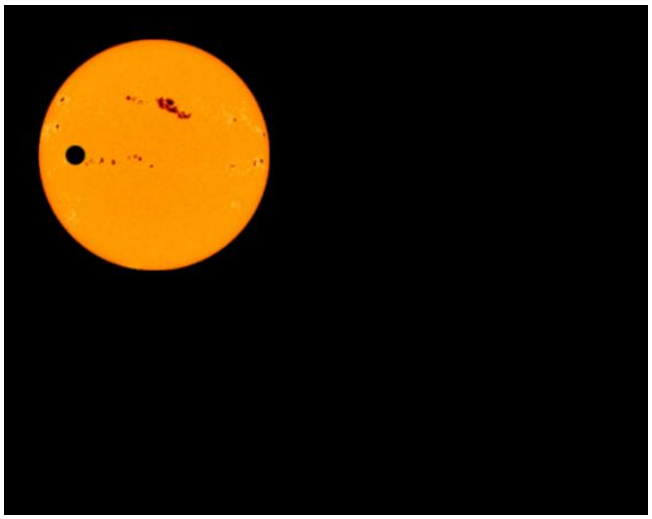


In der *Sensing*-Palette des *ImageSprites* finden sich einige neue Blöcke zum Zugriff auf Werte der Bildpunkte. Der Block ***… by mouse*** wurde schon weiter oben beim Datenimport beschrieben. Mit ***image value … at …*** und ***set image value of …*** können einzelne Pixel gelesen bzw. verändert werden, wenn sie sich im Datenbereich ***myData*** befinden. ***RGB at …*** und ***set RGB at …*** ermöglichen das gleiche direkt auf dem aktuellen Kostüm. Der Block ***brightness around …*** bestimmt die Gesamthelligkeit im Umkreis des angegebenen Punktes.



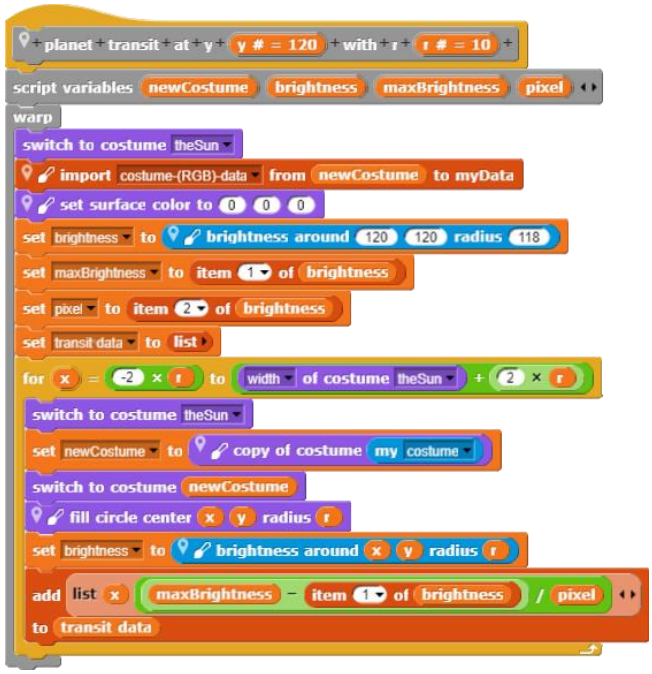
Beispiel: Simulation eines Planeten-Transits vor der Sonne

Wir suchen uns ein schönes Bild der Sonne (Quelle hier: [SchulAstro]) und laden es als Kostüm eines *ImageSprites*. Damit es mehr nach Weltall aussieht, vergrößern wir die Bühne und färben sie schwarz. Zeichnen wir noch den Planeten, dann erhalten wir das nebenstehende Bild.



Der Planet soll als schwarzer Kreis vor der Sonne vorbeiziehen. Wenn wir einen solchen Kreis malen, dann verändern wir das eigentliche Sonnenbild. Von diesem ziehen wir deshalb eine Kopie ***newCostume***, auf der wir dann zeichnen. Unser Planet soll sich von ganz links etwas außerhalb des Bildes (***x***=*-2r*) nach ganz rechts (***x***=*Bildbreite+2r*) auf der Höhe ***y*** bewegen. Auch den Radius ***r*** des Planeten können wir vorgeben.

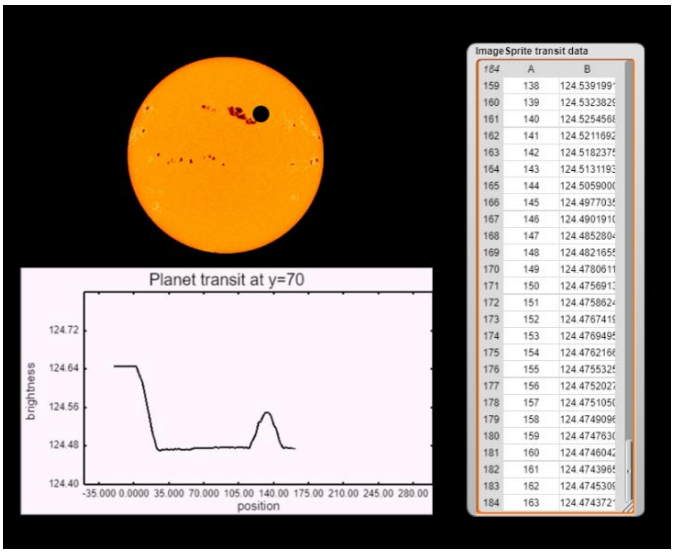
Die aktuelle Helligkeit dieser Anordnung können wir ohne allzu viele Kopiervorgänge bestimmen, indem wir von der anfangs bestimmten Gesamthelligkeit jeweils die Helligkeit der vom Planeten verdeckten Pixel abziehen. Dazu wird anfangs das Bild der Sonne in den Datenbereich ***myData*** importiert und die Helligkeit um den Bild-Mittelpunkt im Radius „halbe Bildbreite“ sowie die Anzahl der beteiligten Pixel bestimmt. ***brightness around*** liefert den summierten Grauwert sowie diese Anzahl. Aus diesen Größen berechnen wir jeweils die mittlere Helligkeit der „leicht verdunkelten“ Sonne und speichern sie zusammen mit der aktuellen Position in der Variablen ***transit data***. Diese Operationen verpacken wir im neuen Block ***planet transit at <y> with r <r>***.



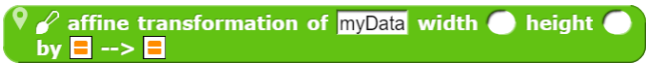
Den Planetentransit wollen wir jetzt „live“ in einem Diagramm verfolgen. Dazu laden wir zusätzlich das *PlotSprite* und schreiben für dieses einen Block zu Vorbereitung der Diagramm-Parameter. Diesen können wir dann vom *ImageSprite* aus aufrufen.

Wir fügen den entsprechenden am Anfang unseres Transit-Skripts ein und ergänzen es durch zwei Aufrufe am Ende jedes Schleifendurchgangs, mit denen das Diagramm des *PlotSprites* mit den neuen Daten neu gezeichnet wird.

Das Ergebnis entspricht in etwa einer der Methoden, mit denen Exo-Planeten gefunden werden.

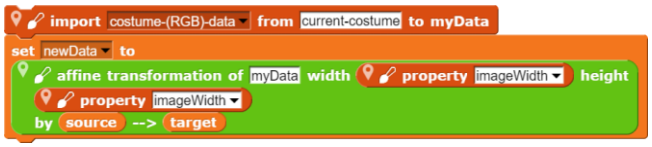


In der *Operators*-Palette finden wir noch das überall benutzte ***is … a …***-Prädikat sowie zwei etwas anspruchsvollere Blöcke. Der erste gestattet es, affine Transformationen in einem Bild vorzunehmen, indem man drei Punkte auf drei andere abbildet – und alle anderen Punkte entsprechend. Zusätzlich müssen die Dimensionen des Bildes angegeben werden.

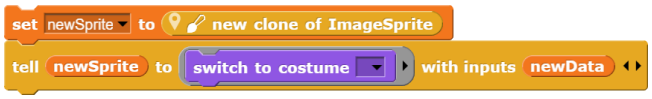


Beispiel: Wir wollen wir ein Bild an der Mittellinie vertikal spiegeln. Wir laden das Bild – hier: einer Kirche – und wählen dazu entsprechende Punkte an den Rändern aus. Diese fassen sie zu den beiden Listen ***source*** und ***target*** zusammen.

Danach importieren wir das Bild in den Datenbereich ***myData*** und führen die affine Transformation mit den beiden Punktlisten aus. Das Ergebnis speichern wir in der Variablen ***newData***.



Zuletzt erzeugen wir einen Klon des *ImageSprites* und bitten diesen, das transformierte Bild als Kostüm anzuzeigen.

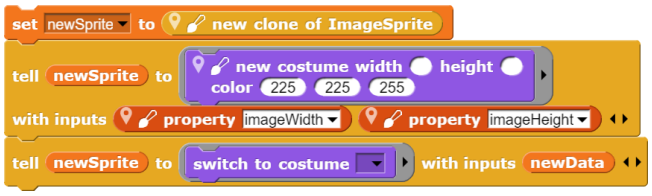


Als letzten neuen Block finden wir den ***apply convolution kernel … to myData*** – Block, mit dem wir Faltungen auf Bildern durchführen können.

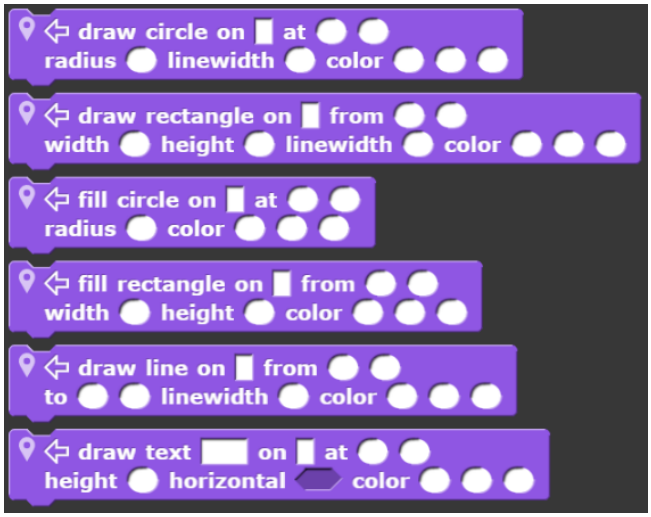
Beispiel: Kantenerkennung

Wir laden ein Bild eines antiken Tempels und importieren seine Daten in den Datenbereich von ***myData***. Auf diese wenden wir den *Laplace*-Kernel $\begin{bmatrix} 1 & 1 & 1 \\ 1 & -4 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ an und speichern das Ergebnis in der Variablen ***newData***.

Danach erzeugen wir einen Klon des *ImageSprites*, passen seine Größe an und lassen ihn das veränderte Bild als sein Kostüm anzeigen.



Da ab und zu Bedarf an der Ausführung der Grafikmethoden im Kontext eines anderen Sprites besteht (s. oben), liegen diese Methoden in einer „exportierbaren“ Version vor, die nicht auf lokale Größen zurückgreift. Sie sind durch einen Linkspfeil ⇦ gekennzeichnet. Die Nutzung ist in Kapitel 4.2 beschrieben.

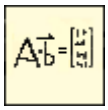


Aufgaben:

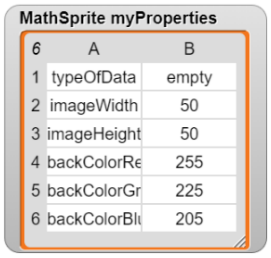
1. Bilder sind manchmal etwas „flau“. Das liegt daran, dass sie nicht den vollen Wertebereich für die drei Farbkanäle von 0 bis 255 ausnutzen.
   a: Entwickeln Sie eine Methode, die Wertebereiche eines Bildes zu ermitteln und anzeigen zu lassen.
   b: Entwickeln Sie eine Methode; den vollen Wertebereich auszuschöpfen, also schwarze Pixel auf 0, helle auf 255 abzubilden.
   c: Fassen Sie das Verfahren in einem neuen Block zusammen, dem das Kostüm eines Sprites übergeben wird und der das verbesserte Kostüm als Ergebnis zurückgibt.

2. a: Auf Bildern kann man versuchen, „Gesichter“ zu finden, indem man zusammenhängende Bereiche eines Farbbereichs, z. B. „orange“, hervorhebt und den Rest des Bildes löscht. Versuchen Sie, dafür einen neuen Block zu entwickeln.
   b: Mithilfe eines Kernels können die Ränder solcher Bereiche isoliert werden. Informieren Sie sich z. B. im Netz über geeignete Kernels und erproben Sie diese bzgl. des genannten Zwecks.
   c: Gesichter sind oft „oval“. Versuchen Sie auf diesem Wege Gesichter von anderen „orangenen“ Gegenständen zu unterscheiden.

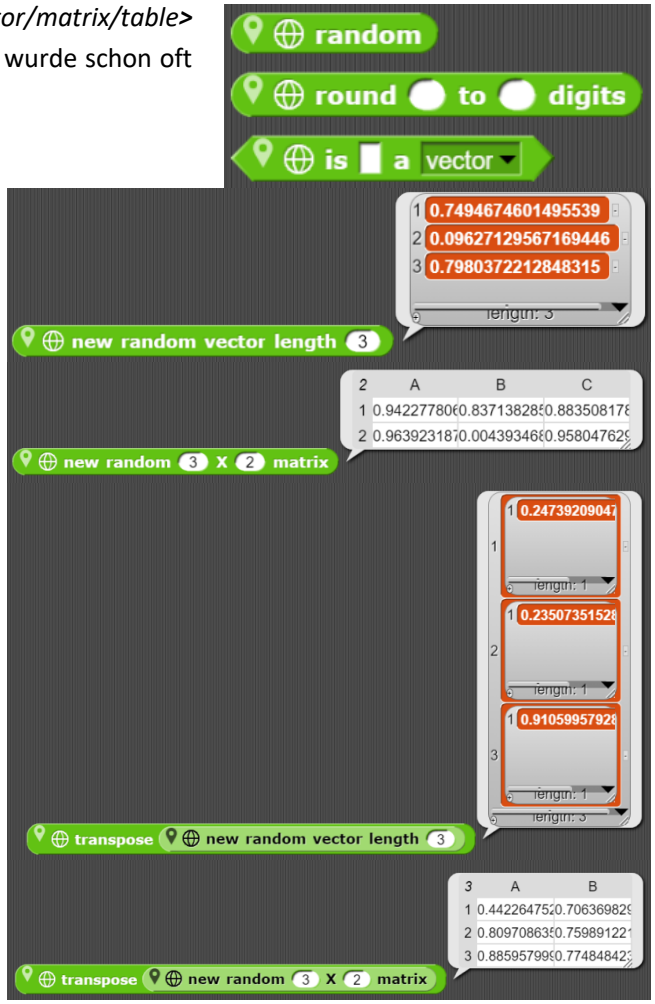
3. a: Richtig künstlerische Fotos sind natürlich schwarz-weiß. Wenn man keine hat, kann man aus RGB-Bildern Graustufenbilder erzeugen. Tun Sie das.
   b: Noch künstlerischer wirkt es, wenn die Fotos „hart“ sind, also einen sehr starken Kontrast haben. Experimentieren Sie mal ein bisschen!

4.7 Das MathSprite 

Das *MathSprite* kann zwar Klone und ein neues Kostüm erzeugen und seine - wenigen - Eigenschaften verwalten; ansonsten hat es aber nur neue Blöcke in der *Operators*-Palette. Es ist für Dienstleistungen für die anderen Sprite-Typen gedacht. Im Wesentlichen ergänzt es die Operatoren von *Snap!* um Möglichkeiten der linearen Algebra.



Drei der Blöcke sind schnell erklärt: ***random*** liefert eine Zufallszahl der üblichen Art, also zwischen 0 und 1 mit voller Genauigkeit. ***round*** rundet Zahlen mit Nachkommateil auf die angegebene Zahl von Stellen. Das Prädikat ***is <data> a <vector/matrix/table>*** wird meist in Skripten zum Fehlerabfangen gebraucht und wurde schon oft in diesem Skript benutzt.

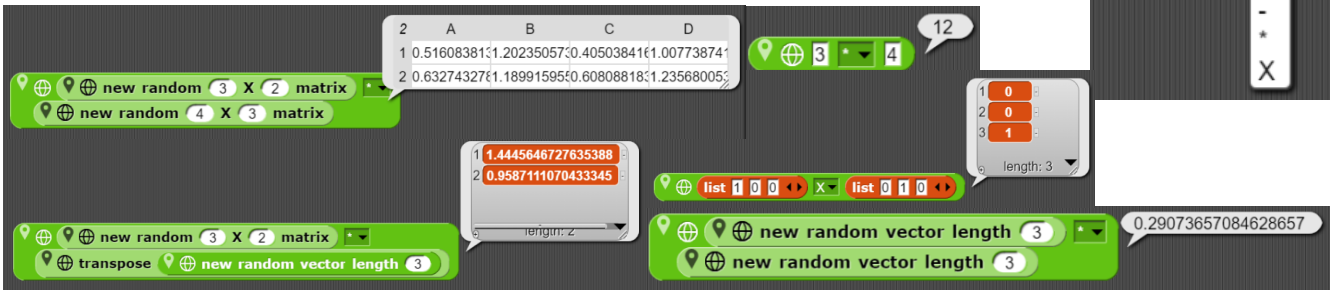


Auch die nächsten beiden Blöcke sollten selbsterklärend sein. ***new random vector*** erzeugt einen neuen Vektor aus Zufallszahlen, den man manchmal zu Testzwecken benötigt. ***new random matrix*** leistet das Gleiche für Matrizen.

Das *MathSprite* arbeitet mit Zahlen, Vektoren und Matrizen. Da Vektoren und Matrizen manchmal in transponierter Form benötigt werden, finden wir auch einen Block ***transpose***. Dieser kann mit beiden Datentypen umgehen.

Interessanter sind die nächsten Blöcke.

Der etwas unscheinbare Reporterblock ***<a> operator <b>*** kann die angegebenen Operation mit Zahlen, Vektoren und Matrizen ausführen – wenn sie zulässig sind. Es klappt also zwischen Zahlen, Zahlen und Vektoren, Vektoren und Matrizen, Matrizen und Matrizen, …



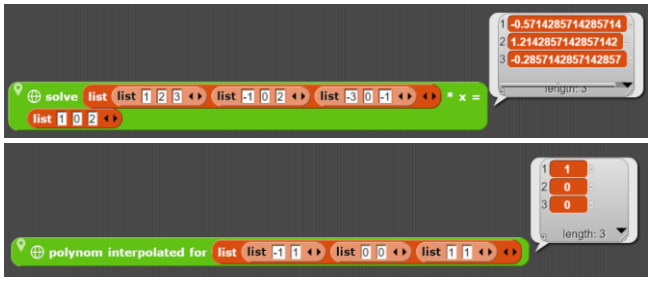
Polynome werden vom *MathSprite* als Listen ihrer Koeffizienten verarbeitet. Der Koeffizient der höchsten Potenz steht links. $x^2 - 2x + 1$ schreibt man 

. Ihren Wert für ein bestimmtes Argument berechnet der Block ***<polynom> polynom (<argument>)***.

***solve** <matrix> * x = <vector>* löst lineare Gleichungssysteme – wenn es eine Lösung gibt.

Gibt man eine Liste von Punkten vor, dann berechnet der Block ***polynom interpolated for** <points>* ein Polynom, dessen Graph durch diese Punkte verläuft.

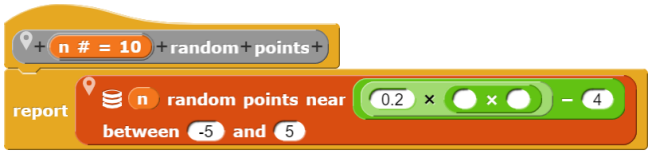
Anwendungen dieser Blöcke folgen.



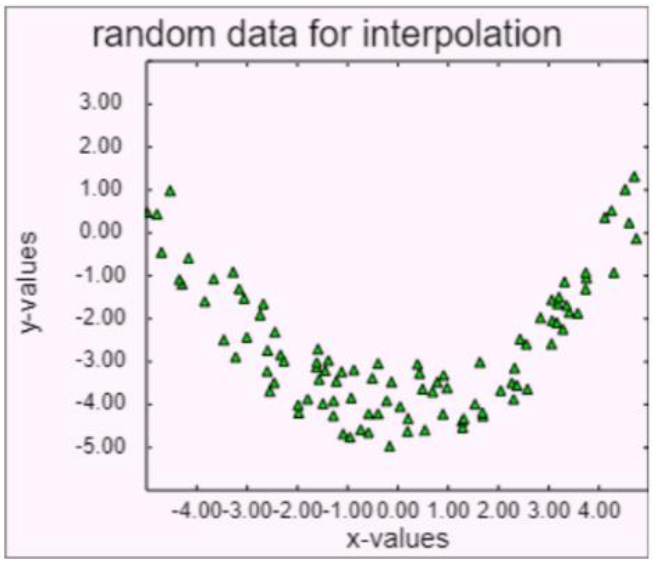
Beispiel: Kurve durch ***n*** Punkte

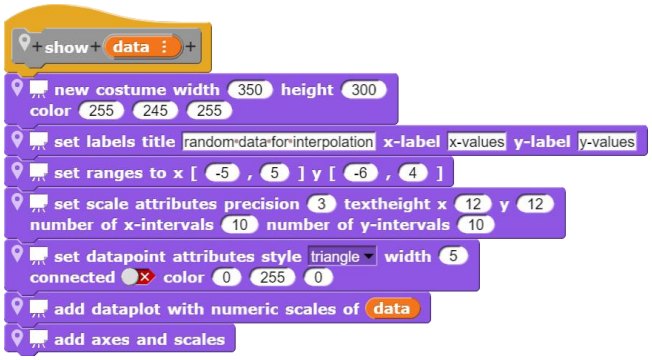
Wir benötigen drei Sprites: ein *DataSprite* zur Erzeugung der Zufallspunkte, ein *MathSprite* zur Berechnung des Interpolations-Polynoms und ein *PlotSprite* zur Darstellung der Ergebnisse. Gesteuert werden soll alles von einem vierten Sprite namens *Control*. Die Punkte für die Interpolation sollen durch Mausklicks ausgewählt werden.

Erzeugung der Zufallsdaten: Im *DataSprite* schreiben wir eine Funktion zur Erzeugung der Punkte. Diese rufen wir von *Control* aus auf.

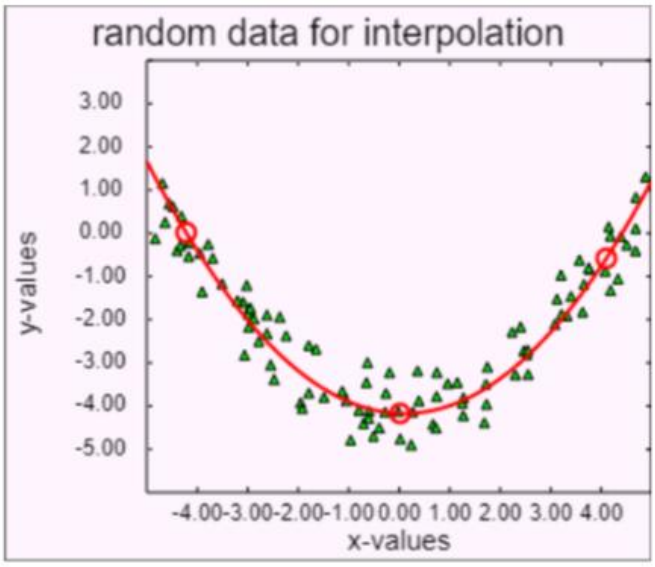


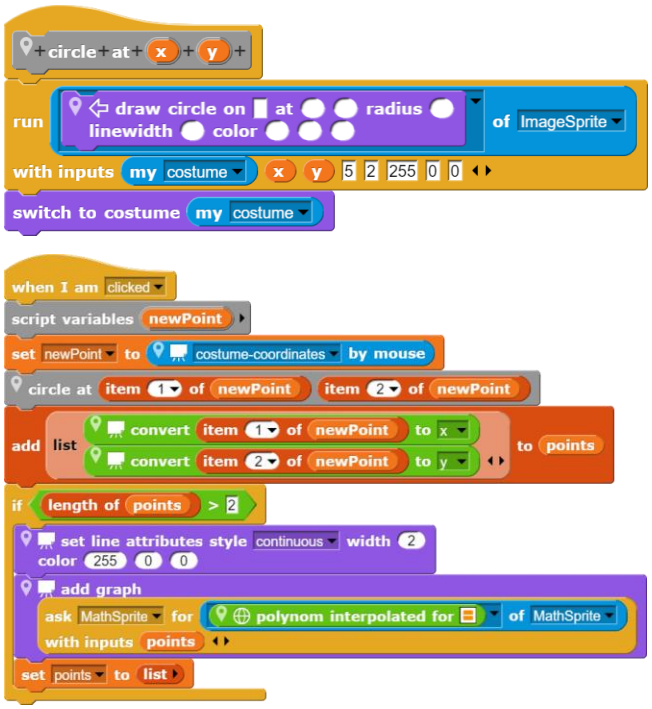
Darstellung der Zufallsdaten: Wir schreiben im *PlotSprite* eine Methode, um die Daten darzustellen. Dieser übergeben wir von *Control* aus die Daten.





Interpolationsdatenaufnahme mit der Maus: Wir ergänzen das *PlotSprite* um eine Ereignisbehandlungsmethode, die auf Mausklicks reagiert. Mit ***<costume-coordinates> by mouse*** erhalten wir die Kostümkoordinaten und speichern sie in der Variablen ***newPoint***. Um die Punkte zu sehen, importieren wir die „exportierbare" ***draw circle*** – Methode des *ImageSprites*. Da sie ziemlich viele Parameter erfordert und wir nur kleine rote Kreise zeichnen wollen, benutzen wir eine Hilfsmethode ***circle at <x><y>***, die nur die aktuellen Koordinaten benötigt – der Rest ist schon ausgefüllt. Anschließend speichern wir die auf Graph-Koordinaten umgerechneten Kostümkoordinaten in der Liste ***points***. Haben wir dreimal geklickt, dann lassen wir das *MathSprite* ein Polynom durch diese Punkte berechnen. Dieses zeichnen wir etwas dicker in Rot ein.





In *Control* bereiten wir vorher alles für die Datenaufnahme vor.

Aufgaben:

1. a: Erzeugen Sie „Punktwolken", die um andere ganzrationale Funktionsgraphen streuen.

   b: Legen Sie wie im Beispiel einige Punkte in diesen Wolken fest, durch die ein Interpolationspolynom gezeichnet werden soll.

   c: Lassen Sie diese Polynome zeichnen.

2. a: Experimentieren Sie mit der Anzahl der ausgewählten Punkte. Werden die Ergebnisse besser, wenn Sie mehr Punkte wählen?

   b: Erzeugen Sie „Punktwolken", die um nicht ganzrationale Funktionsgraphen (trigonometrische, …) streuen. Können Sie auch diese durch Interpolationspolynome beschreiben?

   c: Formulieren Sie eine Regel, wann und wie man Interpolationspolynome sinnvoll einsetzen kann - und weshalb gerade so.

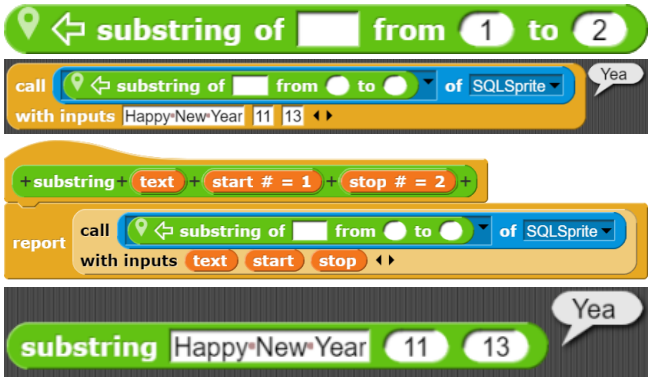
4.8 Das SQLSprite

Ähnlich wie das *MathSprite* ist das *SQLSprite* als Dienstleister für Projekte gedacht. Es kapselt die Funktionalität für Datenbank-Zugriffe und sollte folglich auch nur dann geladen werden, wenn man es braucht.

SQLSprite myProperties

12	A	B
1	typeOfData	empty
2	connected	false
3	connection	https://snapextensions.
4	current-database	
5	current-table	
6	databases	
7	tables	
8	attributes	
9	columns	0
10	rows	0
11	minValue	not set
12	maxValue	not set

Die wenigen Properties des *SQLSprites* enthalten im Wesentlichen die Verbindungsdaten zu einem Beispiel-Datenbank-Server sowie den aktuellen Zustand der Verbindung (genutzte Datenbank, aktuelle Tabelle, …). Man kann zwar Klone des *SQLSprites* erzeugen, aber es wird wenige Projekte geben, bei denen das nötig ist. Auch neue Kostüme braucht man selten. Meist wird das SQL-Sprite nur „irgendwo herumliegen“ und durch die Farbe seines Kostüms anzeigen, ob die Verbindung zum Server steht.



Außerhalb der *Variables*-Palette gibt es nur einen Operator-Block zur Bestimmung des Teils einer Zeichenkette – das benötigt man ab und zu, um die Abfrageergebnisse auszuwerten. Weil keinerlei lokale Daten dafür benötigt werden, ist der Block durch den Linkspfeil als „exportierbar“ gekennzeichnet. Er kann deshalb von allen Sprites unabhängig vom Kontext ausgeführt werden. Passiert das öfter, dann sollte man z. B. eine globale Methode ***substring*** schreiben, um den Aufruf zu erleichtern.

In der *Variables*-Palette findet man dann die eigentlichen SQL-Blöcke. Will man eine Verbindung herstellen, dann kann entweder der vorgegebene Server benutzt oder eine neue Verbindung mit ***set property<connection>*** und den Verbindungsdaten eingestellt werden. Anschließend sollte der Block ***connect*** die Verbindung aufbauen. Klappt das, dann wechselt das *SQLSprite* zum grünen Kostüm. Die zur Verfügung stehenden Datenbanken werden mit ***read databases*** aufgelistet. Eine davon kann mit ***choose database*** ausgewählt werden.

Der Umgang mit Tabellen ist entsprechend. Auch hier kann eine bestimmte ausgewählt und angezeigt werden.

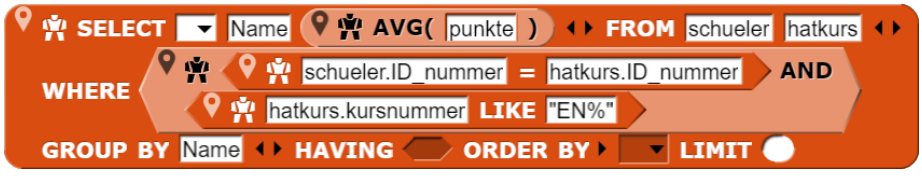
In der Praxis benötigt man dauernd die Details der Tabellen der benutzen Datenbank. Weist man nacheinander die Tabellen-Attribute einer Variablen zu und lässt diese mit *„open in dialog“* anzeigen, dann kann man die entsprechenden Tabellendaten geeignet im *Snap!*-Fenster platzieren und mit den Abfragen beginnen.

SQL-Anfragen werden als SELECT-Anweisungen zusammengestellt. Dafür gibt es zwei Blöcke: einen für einfache, einen für (fast) vollständige Anweisungen.

Mithilfe der Standardprädikate und Funktionen von SQL stellt man die Anfragen zusammen.

Beispiel: eine einfache SELECT-Anfrage

Beispiel: eine komplexere SQL-Anfrage



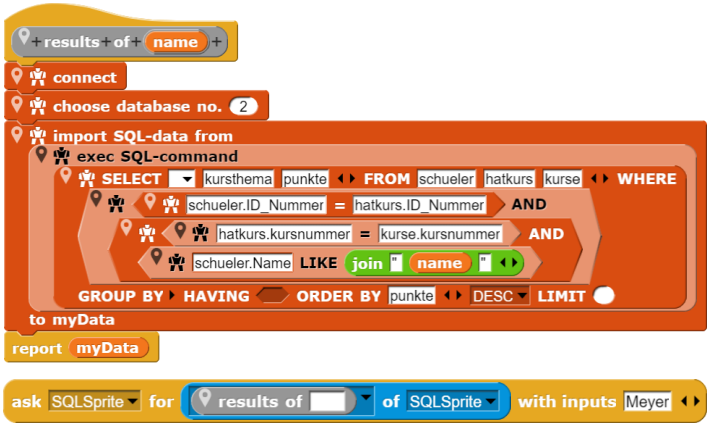
Die beiden SELECT-Blöcke erzeugen den Text einer SQL-Anfrage, führen diese aber nicht aus. Der Grund ist einfach: man soll sich die Anfrage ansehen können. Ist man damit zufrieden, dann wird sie mit ***exec SQL-command*** ausgeführt. In diesen Block kann man auch andere SQL-Befehle eingeben, wenn man die entsprechenden Rechte auf dem Server hat. Mithilfe von ***import SQL-data …*** werden diese dann in Tabellenform überführt und in den Datenbereich des *SQLSprites* verschoben.

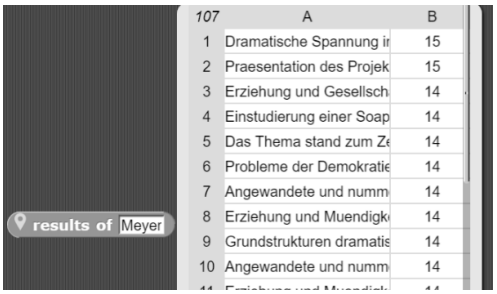
Aus diesen Blöcken lassen sich komplexere Anweisungen zusammensetzen und erproben. Entspricht das Ergebnis den Erwartungen, dann lässt sich die Anfrage in einem Block kapseln, dem nur noch die relevanten Parameter übergeben werden und der dann von anderen Sprites ohne Detailkenntnisse genutzt werden kann.

SQLSprite myData

82	A	B
1	Aehrlich	7.5000
2	Antolni	4.5000
3	Bahn	6.2500
4	Batton	11.2500
5	Benner	10.7500
6	Berg	6.2500
7	Beusberg	7.7500
8	Boemmel	4.2500
9	Brummel	9.2500

Beispiel: Die Kurstitel und Bewertung für alle Kurse eines Lernenden, absteigend sortiert nach der Note, werden gesucht.

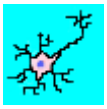




107	A	B
1	Dramatische Spannung i	15
2	Praesentation des Projek	15
3	Erziehung und Gesellsch	14
4	Einstudierung einer Soap	14
5	Das Thema stand zum Ze	14
6	Probleme der Demokratie	14
7	Angewandete und numm	14
8	Erziehung und Muendigk	14
9	Grundstrukturen dramatis	14
10	Angewandete und numm	14

Beispiel: Für statistische Zwecke soll die ***schueler***-Tabelle nach unterschiedlichen Kriterien durchsucht werden können.

Beispiel: Die 10 Kurse mit den „besten“ Ergebnissen sollen absteigend sortiert ausgegeben werden.

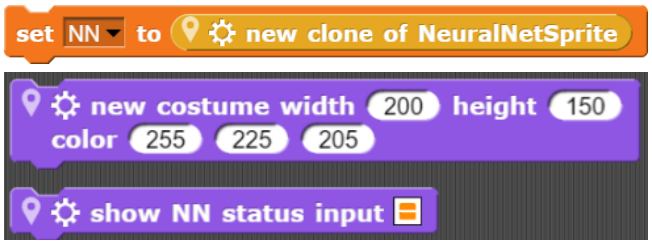
4.9 Das NeuralNetSprite 

„Tiefe“ Neuronale Netze prägen die Diskussion über die aktuelle „künstliche Intelligenz“. Dabei handelt es sich meist um „fully connected“ Netze aus mehreren Perzeptron-Schichten. „Fully connected“ bedeutet, dass alle Neuronen einer Schicht mit allen der nächsten Schicht verbunden sind. Jede Verbindung ist mit einem Gewicht versehen, aus dem sich sein Einfluss auf das verbundene Perzeptron ergibt – aber das lesen Sie besser woanders nach.

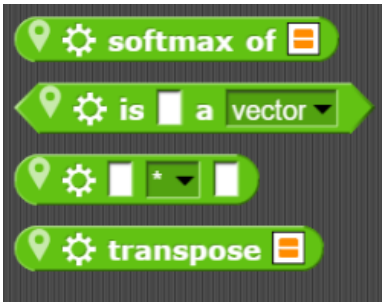
Betrachten wir dazu einmal ein Netz aus drei Lagen, dass als Eingabe die Pixel eines aktuellen 20 M-Pixel-Fotos erhält, also $2x10^7$ Pixel. Die Eingabeschicht besteht aus $3x2x10^7$ MB Zahlenwerten zwischen 0 und 255 (wenn wir das Transparenz-Byte weglassen). Zur nächsten Schicht gibt es dann $(6x10^7)^2=3,6x10^{15}$ Verbindungen – und das dann noch zweimal. Insgesamt wären $3x3,6x10^{15}$, also etwa 10^{16} Gewichte zu bestimmen – eine für „normale“ Rechner völlig utopische Aufgabe. Wir werden uns also auf etwas kleinere Neuronale Netze beschränken müssen.

Eine Möglichkeit, Perzeptron-Netze zu trainieren, besteht darin, ihnen Eingabevektoren zu präsentieren und die gewünschte Ausgabe gleich dazu. Das Netz berechnet dann die Ausgabe, die sich aus den vorhandenen, anfangs zufällig gewählten Gewichten ergibt, und bestimmt die Differenz zum vorgegebenen Ergebnis. Von der letzten Ergebnisschicht ausgehend korrigiert es dann die Gewichte „rückwärts gehend“ so, dass seine Ausgabe „etwas besser“ zum vorgegebenen Ergebnis passt. Das Verfahren nennt sich *Backpropagation*. Auch hierzu sollten Sie sich an anderer Stelle informieren. Aus vielen solcher Korrekturen ergibt sich das trainierte Netz. „Lernen“ bedeutet also, anhand vieler Beispiele die Parameter (die Gewichte) anzupassen. Mithilfe dieser Parameter bestimmt das Netz aus dem Eingabevektor einen Ausgabevektor: es berechnet einen Funktionswert.

Unser *NeuralNetSprite* kann solche Perzeptron-Netze simulieren und trainieren. Dafür können Klone des Netzes erzeugt werden – wie bei den anderen *DataSprites* auch. Für ein *NeuralNetSprite* kann man wie gewohnt Kostüme erzeugen und auf diesen den aktuellen Zustand des Netzes anzeigen. Wir haben damit auch schon die neuen Blöcke in der *Control*- und *Looks*-Palette beschrieben.



Unsere Gewichte bilden insgesamt einen ***Tensor*** mit ***m*** Schichten, die aus ***nxn-Matrizen*** bestehen. *NeuralNetsSprites* sollten deshalb die lineare Algebra beherrschen. Um nicht jedes Mal das *MathSprite* fragen zu müssen, kennt das *NeuralNetSprite* die wichtigsten Operationen selbst. Sie finden sich in der *Operators*-Palette. Neu ist nur die ***Softmax***-Funktion, mit der man z. B. Eingabevektoren skalieren kann. Auch hierzu sollten Sie sich informieren.



Die eigentlichen *NeuralNet*-Blöcke finden wir also in der *Variables*-Palette. Dort werden einerseits die wenigen Eigenschaften des Sprites verwaltet, andererseits kann man den Gewichts-Tensor wie gewohnt laden und als CSV-Datei wieder speichern.

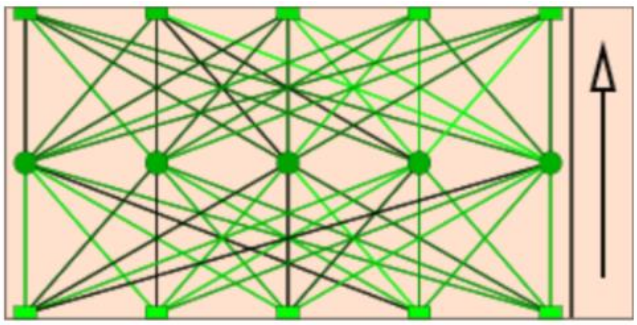
NeuralNet myProperties

8	A	B
1	typeOfData	NN
2	imageWidth	200
3	imageHeight	150
4	backColorRe	255
5	backColorGr	225
6	backColorBl	205
7	layers	2
8	layerWidth	5

Die Dimensionierung und Anfangsbelegung des Netzes erfolgen im Block ***add new weights***. Mit diesen Blöcken können wir also ein neues Neuronales Netz beliebiger Größe erzeugen und anzeigen. In diesem Fall hat es die Breite 5 und die Tiefe 2. Es wird angezeigt, was sich aus den Berechnungen mit dem anzugebenden Eingabevektor ergibt.

Da die Anzeige der vielen Zahlen ziemlich unübersichtlich und auch kaum informativ wäre, werden die Verbindungslinien (die *Kanten*) entsprechend den Werten der zugehörigen Gewichte farbcodiert: von vollem Grün für große positive Werte über Schwarz für kleine Beträge hin zu roten negativen Gewichten. Da anfangs nur positive Zahlen per Zufallsgenerator gezogen werden, ist ein neues Netz überwiegend grün.

Die *Knoten* des Netzes werden wie die Kanten farbkodiert. Unten stehen die Elemente des Eingabevektors als kleine Rechtecke. Die inneren Schichten bilden farbige Kreise und die letzte Schicht wird als Ausgabeschicht wieder rechteckig dargestellt. Die Richtung der Berechnung von unten nach oben zeigt der Pfeil ganz rechts. Da man Sprites aber einfach drehen kann, kann die Richtung natürlich auch anders dargestellt werden.

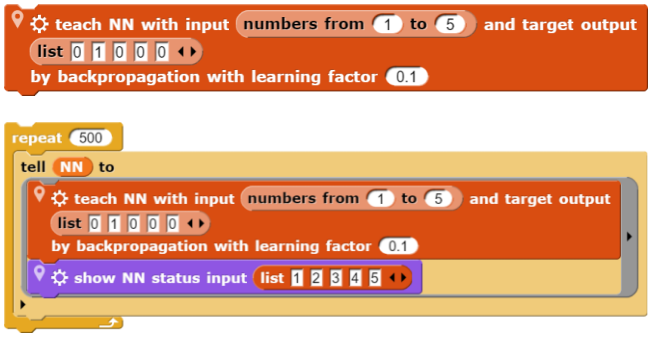


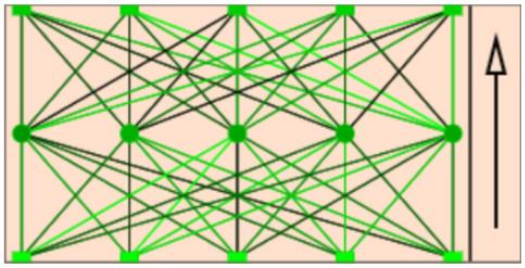
Oft benötigt man die Ergebnisse der letzten oder auch einer inneren Schicht des Netzes. Die können mithilfe des Blocks ***output of…*** bei vorgegebener Eingabe berechnet werden. Da die Farbcodierung nicht unbedingt das größte oder kleinste Element klar anzeigt, kann dieses mithilfe des ***of …***-Blocks bestimmt werden.

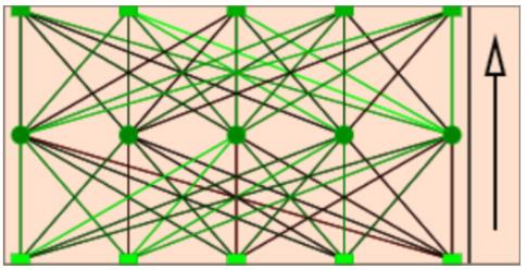
Jetzt müssen wir das Netz trainieren. Im Block ***teach NN …*** erfolgt das durch Backpropagation mit einem anzugebenden Lernfaktor. Der darf anfangs durchaus etwas größer sein, um dann verkleinert zu werden.

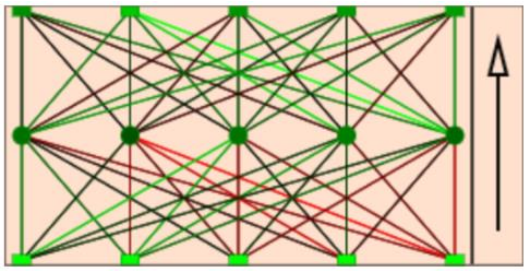
Beispiel: Ein Neuronales Netz soll lernen, aus dem Eingabevektor ***<1,2,3,4,5>*** den Ausgabevektor ***<0,1,0,0,0>*** zu berechnen. Hat es das erfolgreich getan, dann sollte das zweite Ausgabe-Rechteck hellgrün erscheinen, der Rest dunkler.

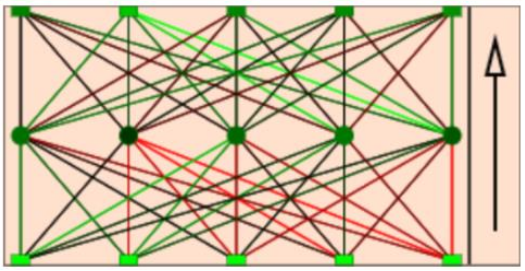
Wir erzeugen also ein neues Neuronales Netz wie oben gezeigt und wenden wiederholt den ***teach***-Block an. Gezeigt wird der Zustand des Netzes nach jeweils 20 Wiederholungen.

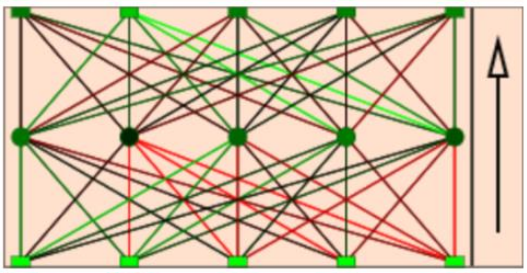


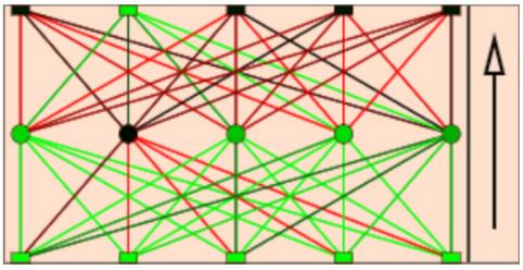












Und nach ca. 200 weiteren Durchgängen hat sich das Muster klar herausgebildet. Das Netz bildet den Eingabevektor auf <0,1,0,0,0> sicher ab. Man kann ganz gut erkennen, dass die erste Schicht die innere Schicht so konfiguriert, dass diese wiederum die Ausgabeneuronen mit der richtigen Gewichtung verstärken (grün) bzw. hemmen (rot) kann.

Natürlich soll das Netz nicht nur ein Muster richtig erkennen. Im Training werden ihm verschiedene weitere Eingabevektoren in zufälliger Reihenfolge mit den gewünschten Ausgaben präsentiert, wobei die Gewichte jeweils leicht so verändert werden, dass das stärkste Ausgabeneuron den richtigen Platz hat. Genügend Parameter dafür hat selbst so ein kleines Netz. Beispiele für Anwendungen finden sich im nächsten Abschnitt.

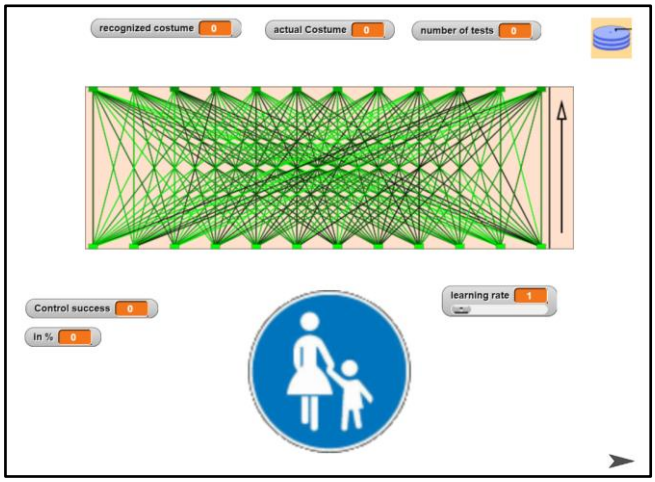
Beispiel: Verkehrszeichenerkennung

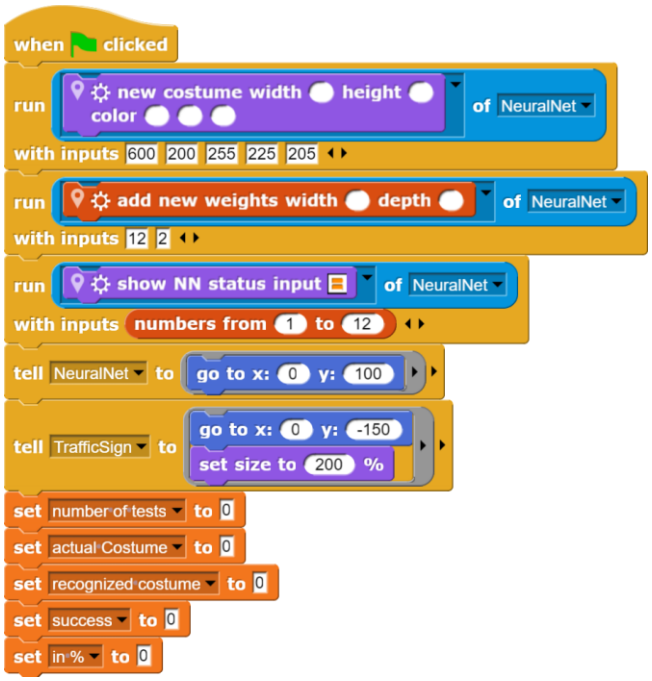
Wir wollen ein neuronales Netz (NN) so trainieren, dass es 12 unterschiedliche Verkehrszeichen erkennt. Dazu suchen wir Bilder von diesen Verkehrszeichen im Netz und verkleinern sie auf das Format 100 x 100 Pixel. Man kann sie jetzt zwar gut am Bildschirm darstellen, aber die 10000 Pixel sind als Eingänge für ein NN natürlich viel zu viel.

Um die Datenmenge in erträgliche Grenzen zu bringen, verkleinern wir die Pixel auf ein 2x2-Format durch ***mean-pooling***, d. h. wir bilden jeweils die Mittelwerte der Farbpixel in den vier Quadranten des Bildes. Aus den 30000 Werten des Verkehrszeichenbildes werden so 12.

Da nur das *DataSprite* die Pooling-Operation beherrscht, importieren wir es neben dem *NeuralNetSprite* zum Projekt mit dem *VerkehrszeichenSprite*. Als viertes Sprite fügen wir ein *ControlSprite* hinzu, das alles steuern soll.

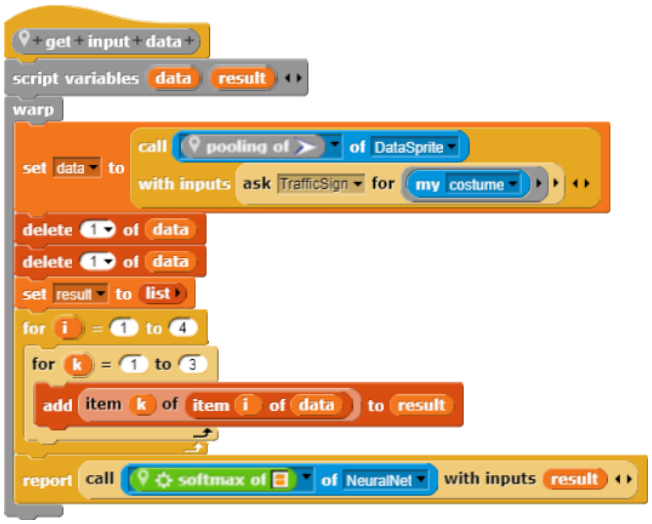
Zum Start verpassen wir dem NN-Sprite ein neues Kostüm. Da wie dafür eine Reihe von Parametern für die Größe und Farbe angeben müssen, benutzen wir den ***run***-Block. Danach erzeugen wir im NN die Gewichte für ein neues (hier: 12x2) Netz. Dieses lassen wir mit einer (noch unsinnigen) Eingabe zeichnen. Danach schicken wir das NN an einen gut gewählten Platz in der oberen Mitte und machen dasselbe mit dem Verkehrszeichen darunter. Zuletzt werden einige Variable auf 0 gesetzt. Die brauchen wir später.



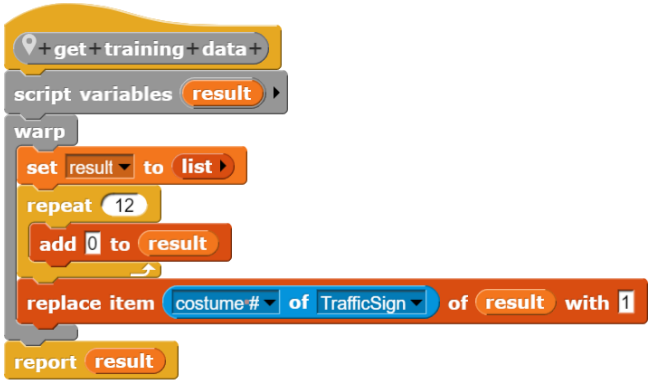


Das *DataSprite* benötigen wir nur zum Rechnen. Um die ***pooling***-Operation anwenden zu können, muss das Sprite die Bilddaten importieren. Danach kann es sie umrechnen und das Ergebnis zurückgeben. Wir fassen alles in einem neuen Block ***pooling of*** *<costume>* zusammen, den wir von *Control* aus aufrufen.

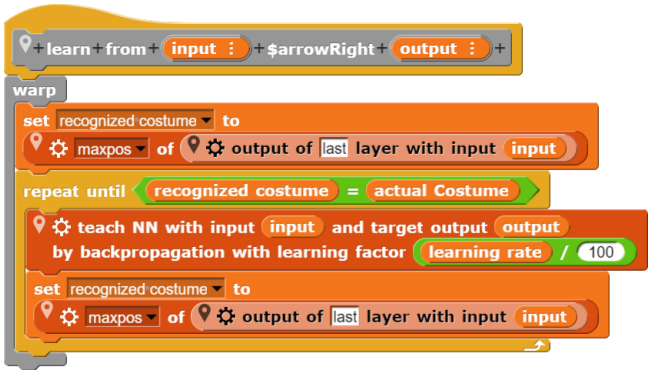
Die Farbwerte des reduzierten Bildes werden mithilfe des neuen Blocks von *Control* zu einem Eingabevektor zusammengesetzt. Die ersten beiden Werte des ***pooling***-Ergebnisses geben die neuen Bilddimensionen an und werden gelöscht. Der Vektor aus ***get input data*** enthält damit die gewünschten 12 Werte. Diese werden abschließend mit der *Softmax*-Funktion des *NeuralNetSprites* modelliert, um ungünstige Eingangswerte auszuschließen.



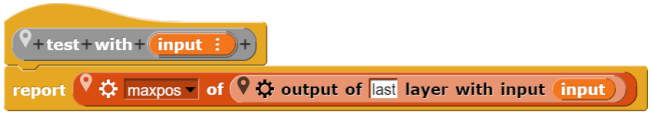
Entsprechend lässt sich der Trainingsvektor in ***get training data*** mit den gesuchten Ausgabewerten des NN ermitteln. In unserem Fall sollen alle Werte 0 sein bis auf den, der der Kostümnummer des Verkehrszeichens entspricht.



Das NN erhält zwei neue Methoden ***learn from…*** und ***test with…*** Beim Lernen wird die Position des Platzes mit dem größten Ausgabewert des NN ermittelt und mit der aktuellen Kostümnummer des Verkehrszeichens verglichen. Stimmen diese Werte nicht überein, dann wird weiter gelernt.

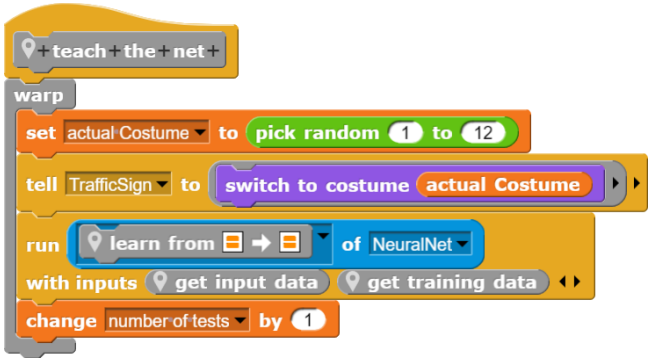


Zum Testen wird dieselbe Operation nur einmal ausgeführt.

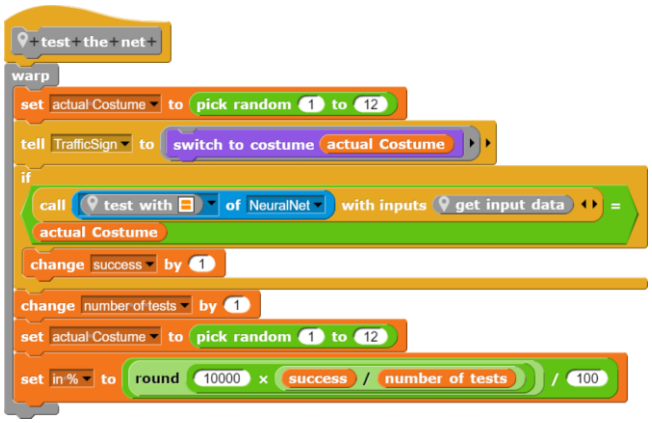


Jetzt haben wir alles zusammen, um *Control* vernünftig arbeiten zu lassen.

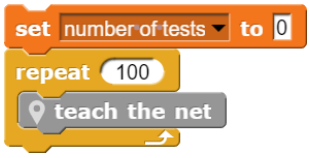
Ein Lehrvorgang besteht aus der Ermittlung einer zufälligen Kostümnummer mit dem entsprechenden Kostümwechsel. Danach wird der Lernvorgang des NN mit neuen Eingabe- und Zieldaten gestartet. Die Durchgänge werden mitgezählt.

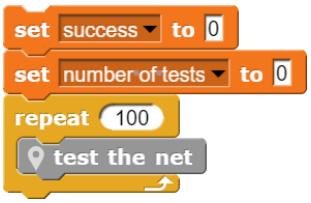


Zum Testen wird ähnlich vorgegangen: Das Kostüm wird gewechselt und geprüft, ob das NN die richtige Kostümnummer berechnet. Ist das der Fall, dann freuen sich alle. Der Prozentsatz der richtigen Versuche wird danach ermittelt.

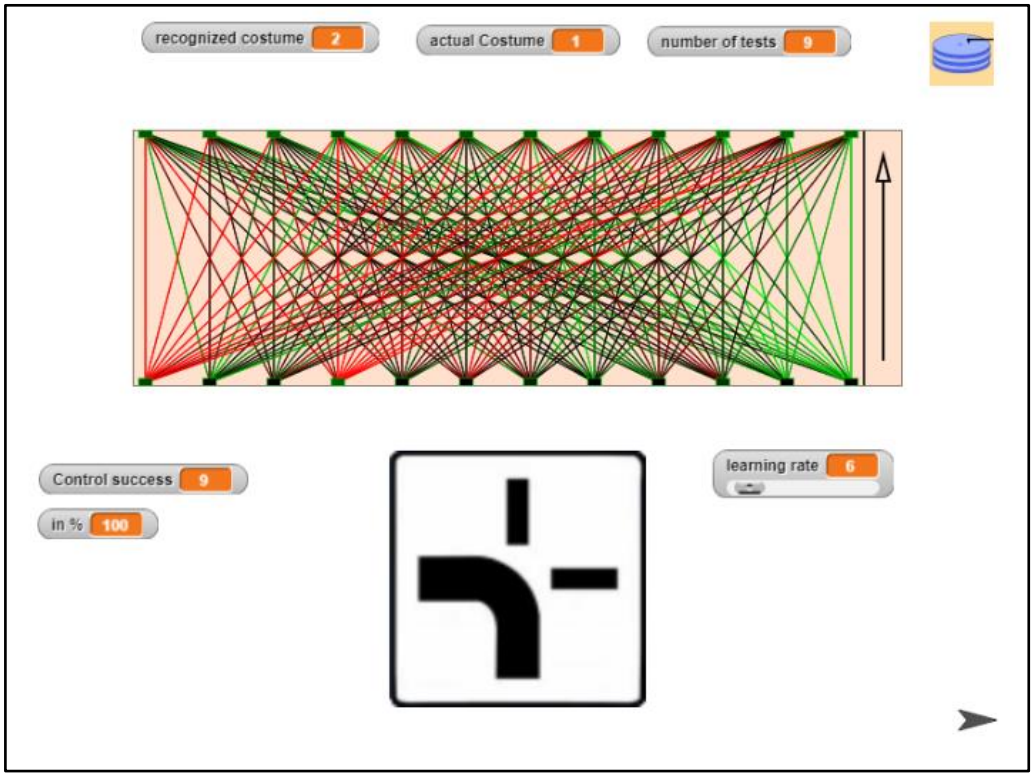


Mehrere Lern- und Testläufe lassen sich dann leicht auslösen.





Nach ca. 50 Trainingsläufen mit einer höheren Lernrate und nochmal 50 mit einer geringen zur Feinabstimmung erreichen wir Erkennungsraten von 100%.



Aufgaben:

1. Trainieren Sie ein einschichtiges Netz mit unterschiedlichen Lernraten und Zahlen der Lerndurchläufe. Ermitteln Sie jeweils prozentual die Erkennungsrate.
2. Stellen Sie die Ergebnisse aus 1. grafisch mithilfe eines *PlotSprites* dar.
3. Experimentieren Sie mit mehrschichtigen NNs. Werden die Ergebnisse besser?
4. Vergrößern Sie die Länge des Eingabevektors durch verändertes Pooling. Werden die Ergebnisse besser?
5. Vergrößern Sie die Anzahl der erkennbaren Schilder, indem Sie mehr als eine 1 in der Ausgabe zulassen.

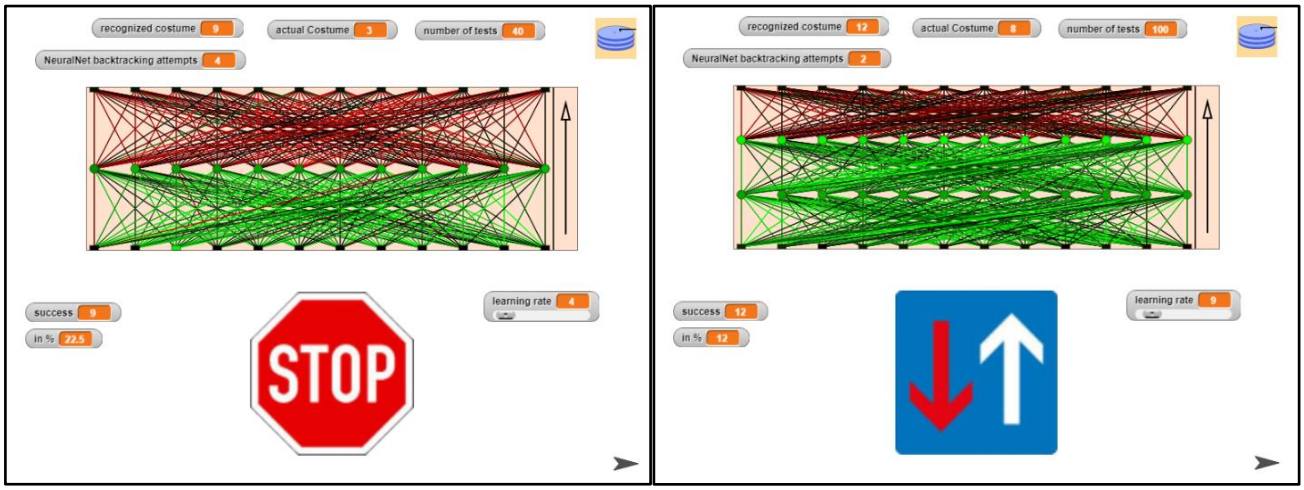
5 Anwendungen der ML.Sprites

5.1 Under- and Overfitting

Maschinelles Lernen passt die Parameter einer Funktion mithilfe von Trainingsdaten so an, dass andere Werte gut prognostiziert werden – wenn alles klappt. Man baut also ein Prognoseinstrument, so eine Art „Fernrohr“ für Daten.

Man könnte nun meinen, dass so eine Funktion desto besser ist, je mehr anpassbare Parameter sie enthält. Dem ist aber nicht so. Einerseits erfordern (1.) mehr Parameter auch mehr Trainingsdaten und Trainingsläufe, also mehr Lernzeit; anderseits kann auch (2.) eine „unpassende“ Zahl der Parameter „gute“ Lösungen verhindern. Für beides geben wir jetzt ein Beispiel.

Zu 1: Beim Neuronalen Netz zur Verkehrszeichenerkennung haben wir mit einer Schicht sehr gute Ergebnisse erzielt. Erhöhen wir die Zahl der Schichten und lassen die Zahl der Trainingsläufe gleich, dann verschlechtert sich drastisch die Erkennungsrate.



Zu 2: Wenn die Trainingsdaten von der Funktion gut reproduziert werden, dann heißt das noch lange nicht, dass das auch für andere Daten gilt. Es hängt sehr von der Art der Funktion ab, die erzeugt wird. Als Anwendung wählen wir das Beispiel *Polynominterpolation*.

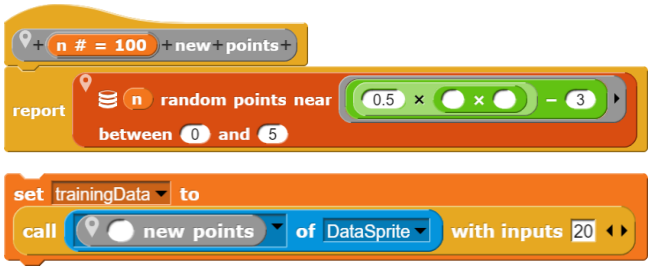
Die Aufgabe lautet: *Mithilfe von Trainingsdaten werden die Koeffizienten eines Polynoms so angepasst, dass ANDERE Daten möglichst gut prognostiziert werden.*

Dafür müssen wir Daten erzeugen, mit deren Hilfe ein Interpolations-Polynom berechnet wird. Die Funktionalitäten dafür verteilen sich auf drei Prototypen: das *PlotSprite* zur Darstellung der Graphen, das *MathSprite* für die Polynominterpolation und das *DataSprite* zur Erzeugung von Zufallsdaten, die um eine vorgegebene Funktion streuen.

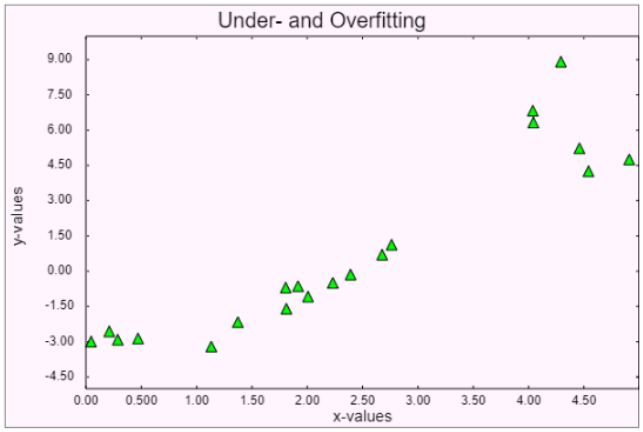
Wir erzeugen also ein neues Projekt, vergrößern die Bühne auf 800x600 Pixel und laden die drei Prototypen. Danach erzeugen wir eine Variable ***random data***. Die Anordnung sieht dann wie nebenstehen aus.

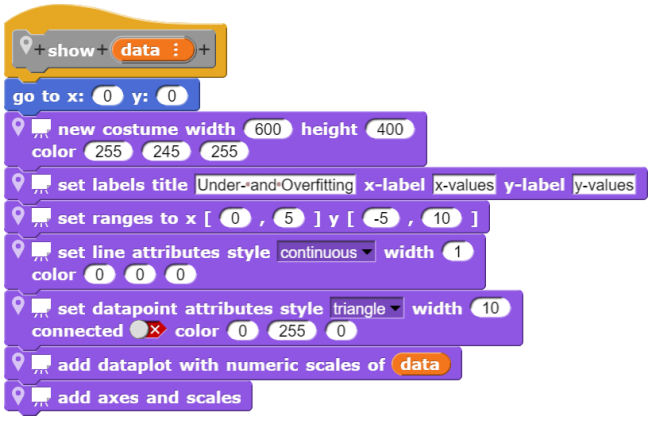
Als „Arbeitspferd“ wählen wir das *PlotSprite*. Benötigte Funktionalität aus anderen Sprites importieren wir bei Bedarf von dort.

Zuerst einmal bitten wir das *DataSprite*, 20 Zufallsdaten zu erzeugen, die um die Parabel $0{,}5 * x^2 - 3$ streuen. Dafür schreiben wir im *DataSprite* eine Funktion ***<n> new points***. Diese rufen wir vom *PlotSprite* aus auf.

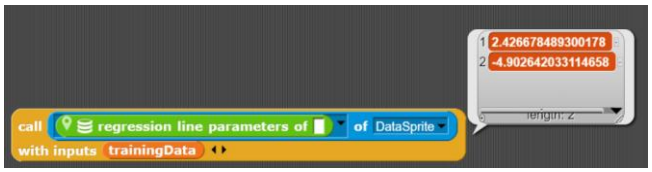


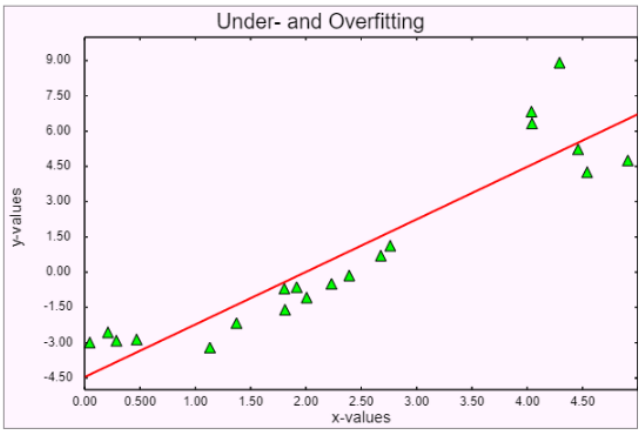
Das genügt schon, um die Daten darzustellen. Wir schreiben einen Block ***show <data>***, der die erforderlichen Einstellungen vornimmt und das erledigt.

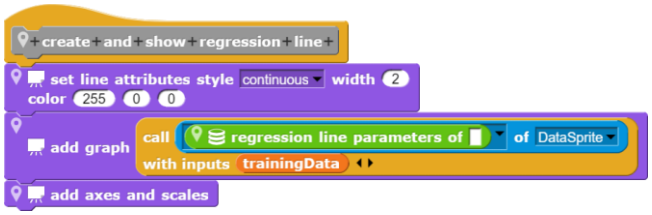




Die Interpolation probieren wir erst einmal mit einer Regressionsgraden. Die erforderlichen Geradenparameter kann das *DataSprite* berechnen.



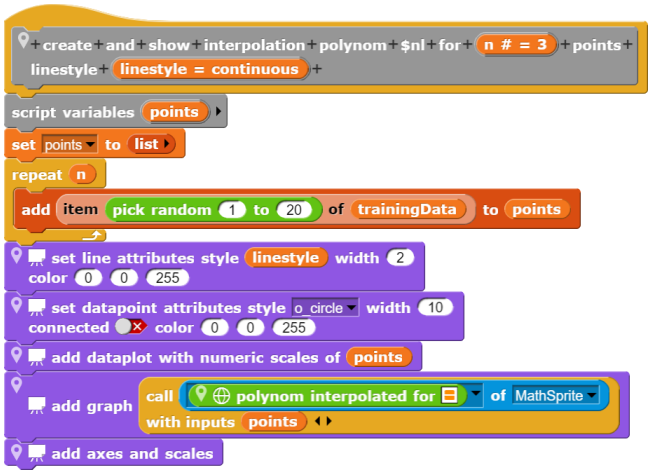


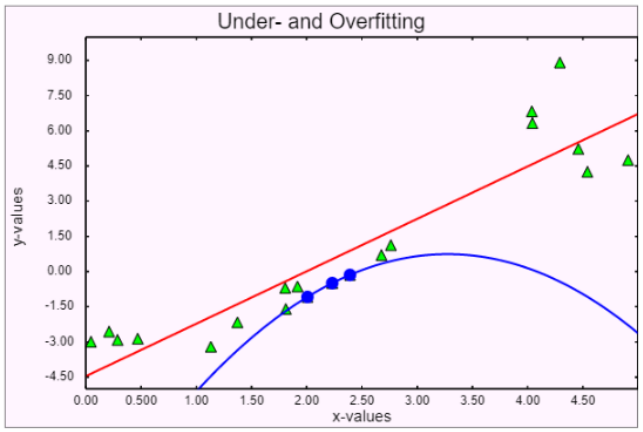


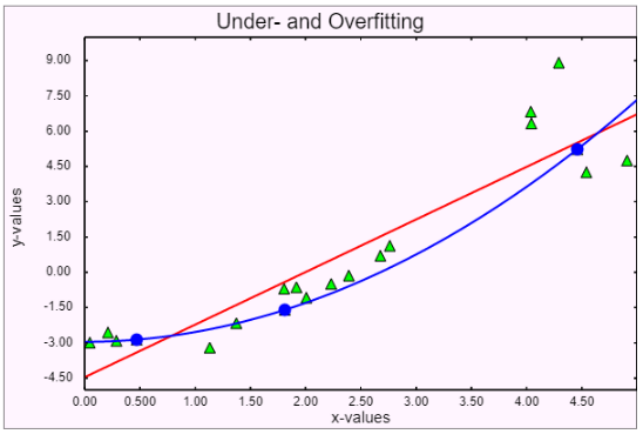
Das sieht eigentlich ganz nett aus, aber an den Seiten passt es nicht so recht.

Wir probieren es also mit einer Polynom-Interpolation.

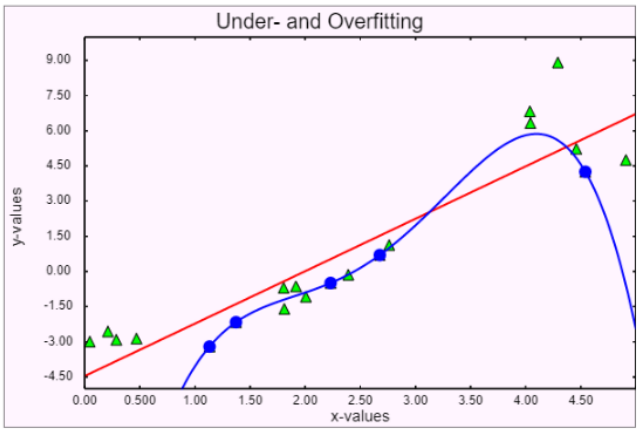
Zuerst einmal wählen wir drei Zufallspaare aus den Trainingsdaten, bestimmen daraus das Interpolationspolynom und zeichnen es. Weil wir weiter experimentieren wollen, verallgemeinern wir die Lösung gleich zu einem Polynom durch ***n*** Punkte. Dabei hoffen wir, dass bei der Auswahl alles gut geht! Die Ergebnisse hängen davon ab, welche Punkte erwischt wurden. Anbei ein schlechtes und ein ganz gutes Ergebnis.



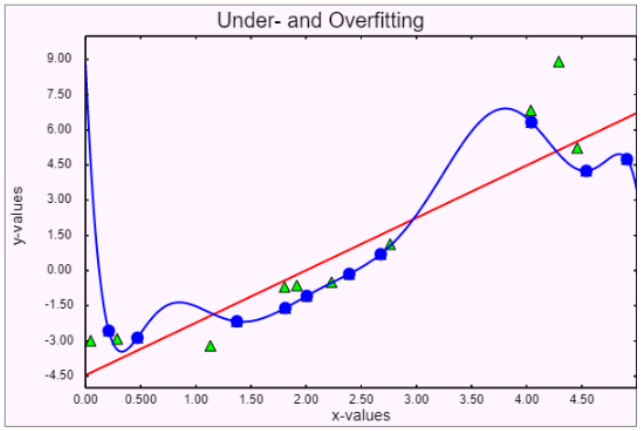


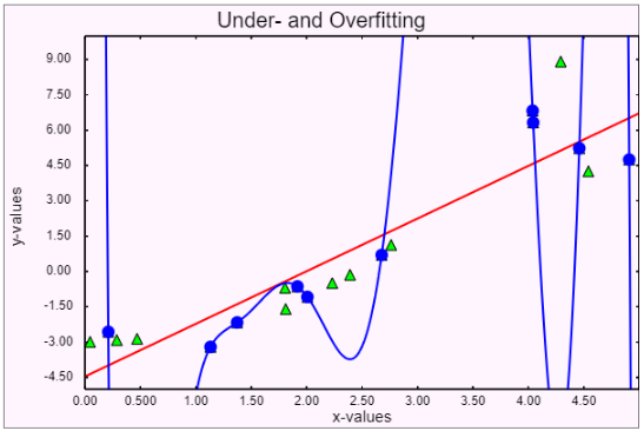


Jetzt werden wir mutig! Statt drei Punkten wählen wir 5. Wir wollen ja schließlich gute Arbeit abliefern! Das klappt prima in der Mitte, und dann – upps!



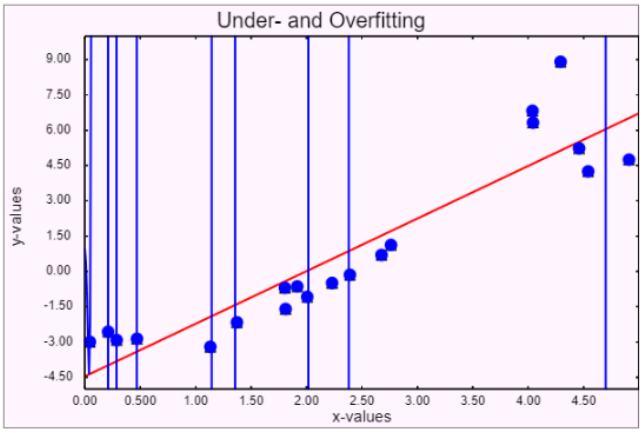
Vielleicht müssen wir ja einfach mehr Punkte nehmen. Versuchen wir es mit 10. Die Polynome verlaufen zwar durch mehr Punkte, aber an den Rändern „hauen sie ab“ – und dazwischen meist auch.





Na, dann mit allen Punkten!

Man sieht, dass mit zunehmendem Grad des Polynoms zwar mehr Trainingsdaten direkt auf dem Graph liegen, dass aber dazwischen durch die wilden Oszillationen des Polynoms nur noch unsinnige Werte „prognostiziert“ werden.



Die Qualität des Gelernten hängt also sehr davon ab, wie wir mit Abweichungen umgehen. Wir müssen entscheiden, welche Ungenauigkeiten im Detail tolerierbar sind, damit die Prognose insgesamt zuverlässig wird. Ist der Grad des Polynoms zu klein, dann spricht man von *Underfitting*, ist er zu hoch, von *Overfitting*.

Aufgaben:

1. Diskutieren Sie unterschiedliche Möglichkeiten, einen „guten“ Grad des Interpolationspolynom (also seine höchste Potenz) festzulegen.
2. Formulieren Sie ihre Ergebnisse so präzise, dass sie sich als Skripte realisieren lassen.
3. Testen Sie die Skripte an unterschiedlichen Datensätzen.

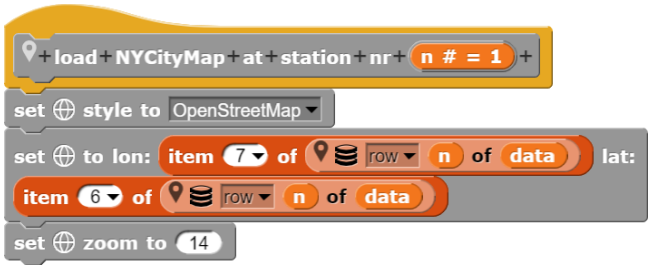
5.2 New York Citibike Tripdata [NYcitibike]

Selbst in New York ist das Fahrradfahren inzwischen „hip“ geworden und die Entleihdaten können als CSV-Dateien geladen werden. Wir tun das und laden die knapp 600 000 Datensätze vom 30. Juni 2013 in eine Tabelle. Dabei spalten wir gleich die Spaltenüberschriften ab, um eine reine Datentabelle zu erhalten. Das geschieht natürlich in einem *DataSprite*. Da wir auch noch Grafiken erstellen wollen, laden wir das *PlotSprite* gleich dazu.

Was haben wir da eigentlich gefunden?

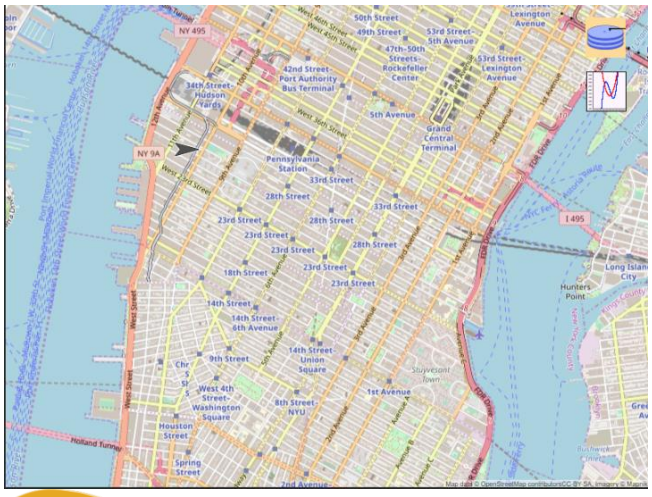
Die Datenlegende liefert die Interpretationsvorschrift für die Daten: *Trip Duration (seconds), Start Time and Date, Stop Time and Date, Start Station Name, End Station Name, Station ID, Station Lat/Long, Bike ID, User Type (Customer = 24-hour pass or 3-day pass user; Subscriber = Annual Member), Gender (Zero=unknown; 1=male; 2=female), Year of Birth*

Da geografische Länge und Breite der Entleihstationen angegeben sind, bietet es sich an, die *World Map Library* von *Snap!* einzusetzen. Wir schreiben dafür einen kleinen Block, der die Umgebung einer Entleihstation als Karte darstellt.



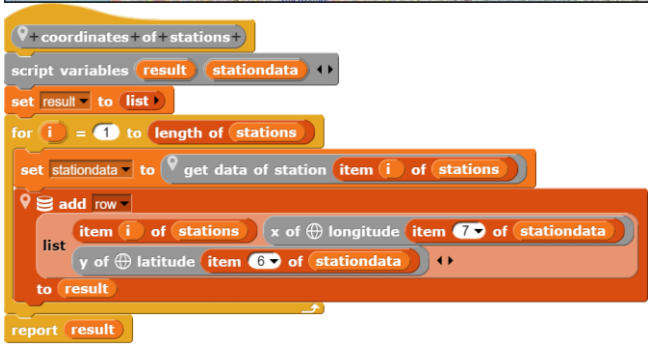
Wir wollen doch mal sehen, wo man Fahrräder entleihen kann. Für die Übersicht extrahieren wir die Entleihstationen aus der Gesamtliste, z. B. indem wir sie nach dem Namen der Startstation (Spalte 5) gruppieren lassen und nur diese Spalte als Ergebnis wählen.

Wir erhalten immerhin 337 Stationen.

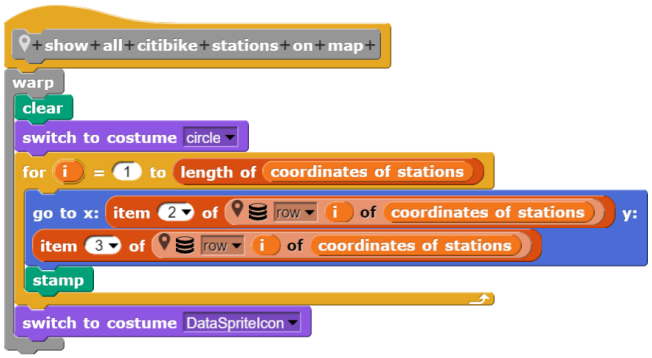


Danach suchen wir die Daten einer Station zusammen …

… und stellen so die Koordinatenliste der Stationen auf.

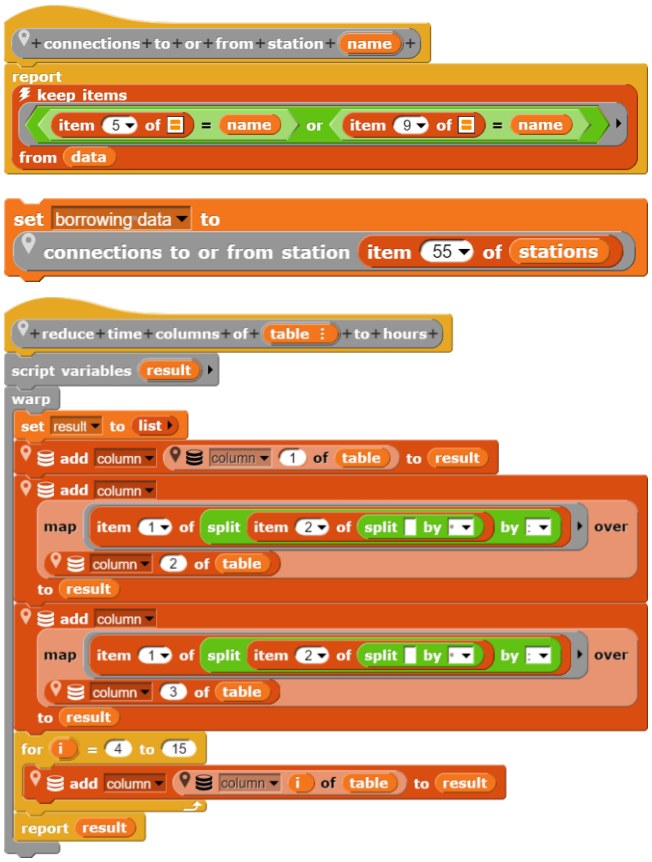


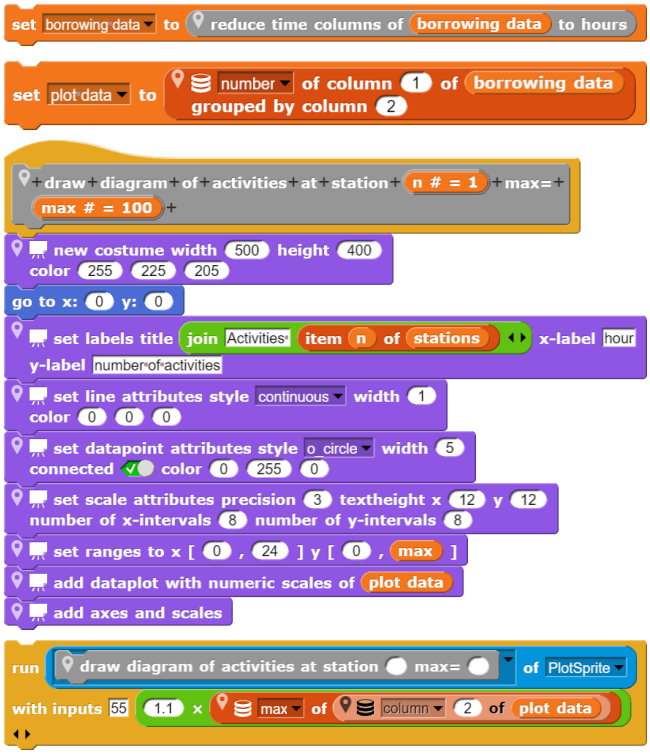
Mit diesen Daten können wir das Sprite zu den einzelnen Positionen schicken und dort mit dem *stamp*-Block zum Hinterlassen von Kreisen auffordern.

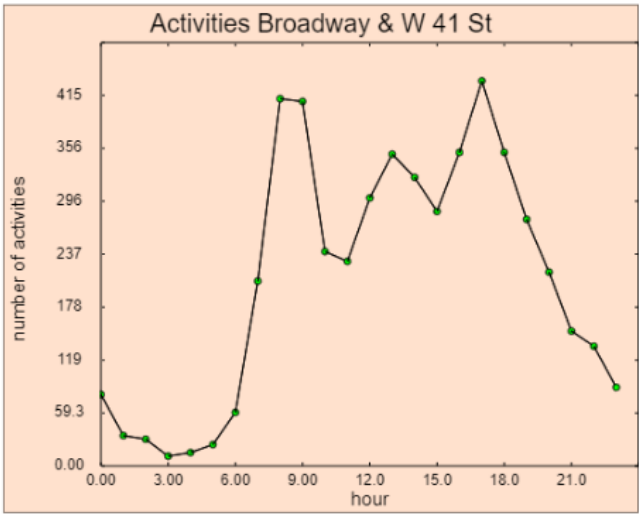


Zumindest in Midtown Manhattan müssen wir uns wohl keine Sorge darum machen, ob wir eine Entleihstation finden!

Wir wollen uns jetzt einmal die Verleihstation Broadway – Ecke 41 Street (Nr. 55) genauer ansehen. Dazu ziehen wir alle Datensätze aus der Gesamtliste, die an dieser Station starten oder enden. Das sind in diesem Monat 5005 Vorgänge. Zeiten sind in dieser Liste zusammen mit dem Datum eingetragen. Das können wir herauswerfen (Abtrennen mit *split* mit „ „) und auf die Stunde reduzieren (*split* mit „:“). Wir haben danach eine numerische Skala mit der Einheit „Stunde“. Jetzt können wir sehen, was in den einzelnen Stunden des Tages an der Station los ist. Und das können wir wie üblich mithilfe des *PlotSprites* grafisch darstellen.

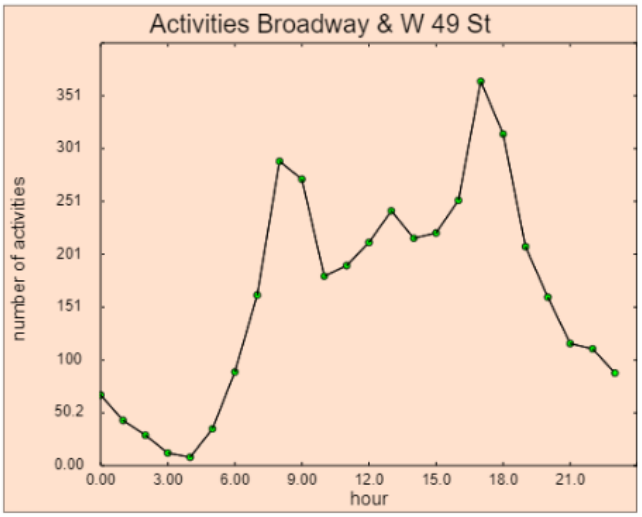




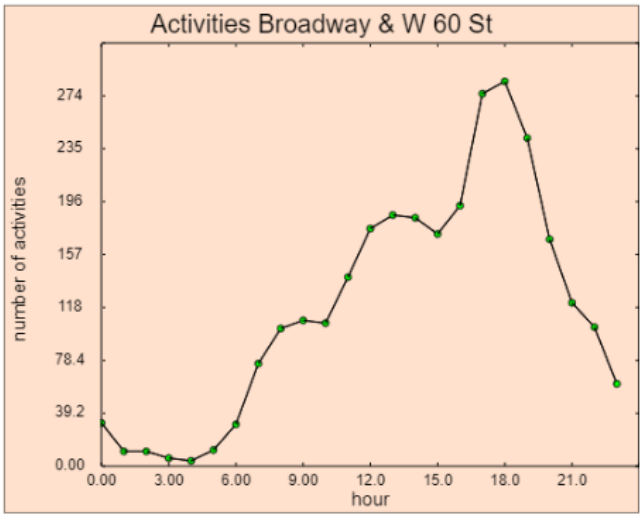


Ein paar Straßen weiter sieht es ganz ähnlich aus.

Ist das ein allgemeines Muster?



Nun gut, am Central Park stehen die Leute später auf und die Touristen sind noch nicht da. Dafür schließen die Museen immer zur gleichen Zeit.



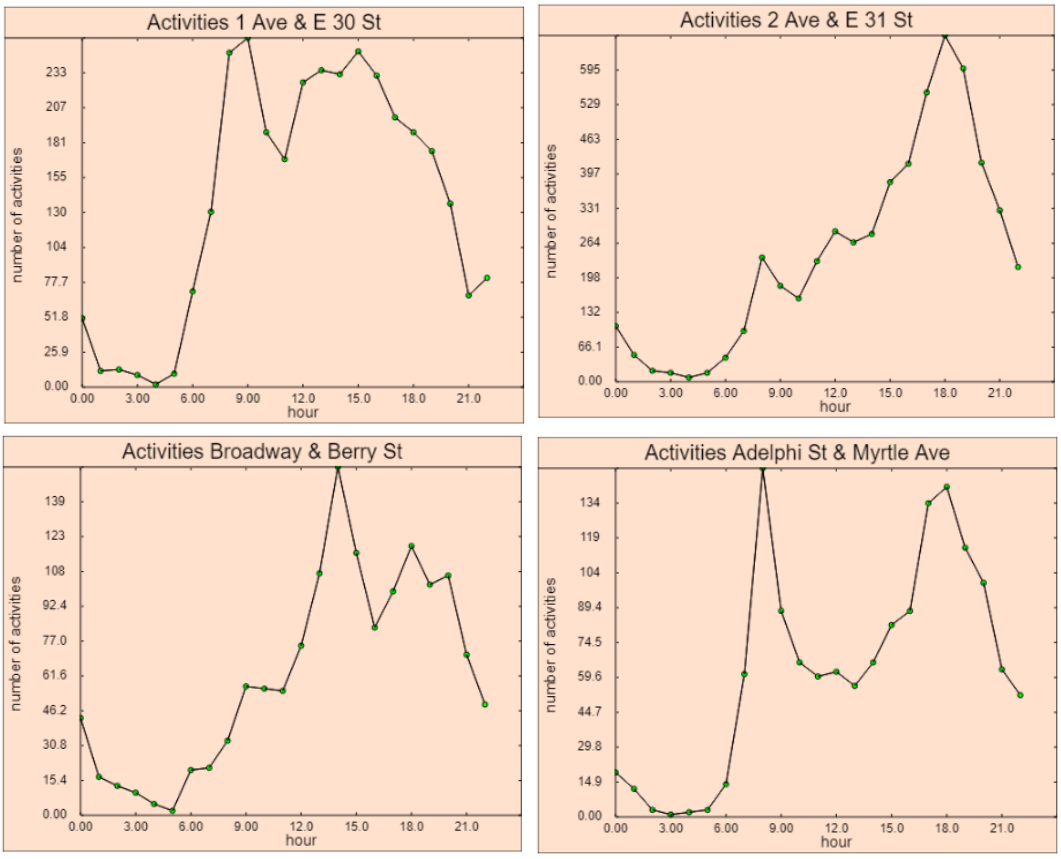
Was können unsere Programme nun aus diesen Daten lernen?

- *Wir könnten z. B. aus den üblichen Ab- und Zugängen sowie dem Istbestand voraussagen, ob an einer Station noch rechtzeitig genügend Fahrräder zurückgegeben werden oder ob es besser wäre, einige dorthin zu transportieren.*
- *Wir könnten aus den mittleren Weglängen ermitteln, welche Akkus für eBikes gebraucht werden.*
- *Wir könnten feststellen, ob eher Frauen oder Männer zu einer bestimmten Tageszeit die Räder entleihen und dann dafür sorgen, dass das Angebot stimmt. Das Entsprechende könnten wir für das Alter der Entleihenden machen.*
- *Wir könnten die Entleihdaten pro Rad ermitteln und voraussagen, wann Reparaturen fällig sein werden. Wir könnten das z. B. auch in Abhängigkeit von der Lage der Stationen machen.*
- *Wir könnten versuchen, Verteilungen von einigen Stationen so zu verallgemeinern, dass sich Prognosen für andere daraus ableiten lassen. Wenn also am Central Park die Museen schließen, kann das Programm aus den alten Daten „lernen“, in welchen Bezirken die Räder wann vermutlich abgegeben werden, und warnen, falls dort nicht genug freie Slots zur Verfügung stehen.*

usw.

Aufgaben:

1. Gliedern Sie die Aktivitäten der Stationen nach Zu- und Abgängen auf.
2. Schreiben Sie eine Prognosefunktion, die warnt, wenn in den nächsten Stunden ein Radmangel an einer Station droht.
3. Stellen Sie für bestimmte Stationen durch direkte Linien die Verbindungen zu den meist gewählten Abgabestationen graphisch auf der Karte dar. Wählen Sie die Liniendicke entsprechend der Anzahl der Entleihvorgängen und die Farben je nach Station. Bilden sich Cluster?
4. Stellen Sie mithilfe von Korrelationen fest, ob es Zusammenhänge im Entleihverhalten (z. B. bzgl. der Tageszeiten, dem Ort, …) mit dem Geschlecht, dem Alter, dem Status der Entleihenden gibt. Ggf. müssen Sie die Daten vorher durch numerische Daten ersetzen – ähnlich wie bei den Zeiten. Diskutieren Sie mögliche Konsequenzen.
5. Ermitteln Sie für einen kleinen Abschnitt von Midtown (dort, wo alles schön rechteckig ist) die Koordinaten der Straßenecken. Entwickeln Sie dann einen Router, der den kürzesten Weg zur nächsten Citibike-Station anzeigt.
6. Die Entleihzahlen abhängig von der Tageszeit zeigen in unterschiedlichen Bereichen Manhattans ziemliche Unterschiede. Untersuchen Sie Gemeinsamkeiten und Unterschiede systematisch und versuchen Sie, die Ergebnisse zu erklären.

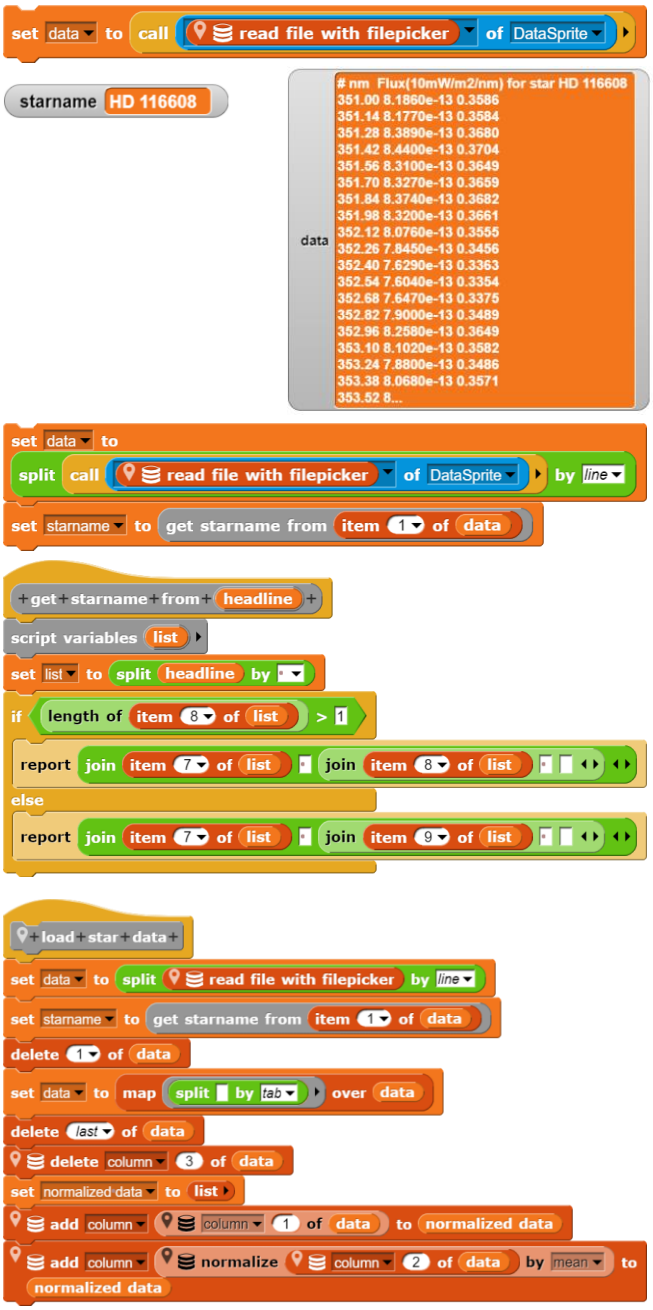


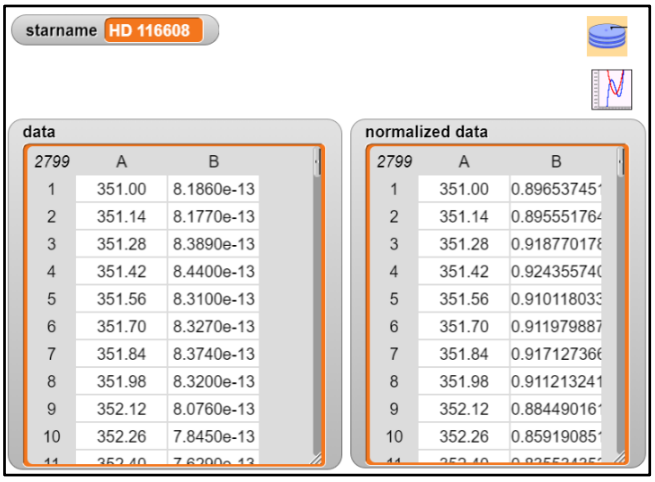
5.3 Sternspektren [UniGOE]

Sterne leuchten in unterschiedlichen Farben, weil sie unterschiedliche Temperaturen haben. Zusätzlich unterscheiden sich die Spektren in ihren Absorptionslinien. Das wollen wir etwas genauer untersuchen. Wir benutzen ein *PlotSprite* und dazu ein *DataSprite* als Hilfe, z. B. zum Laden und Aufbereiten der Daten. In diesem beginnen wir mit der Arbeit.

Wir besorgen uns einige Sternspektren (Quelle: [UniGOE]) und speichern sie als Textdatei. Solch eine lesen wir ein und zerlegen sie gleich in eine Liste ***data***. In der ersten Zeile steht nach den Spaltenbeschriftungen der Sternname. Den isolieren wir und speichern ihn als ***starname***.

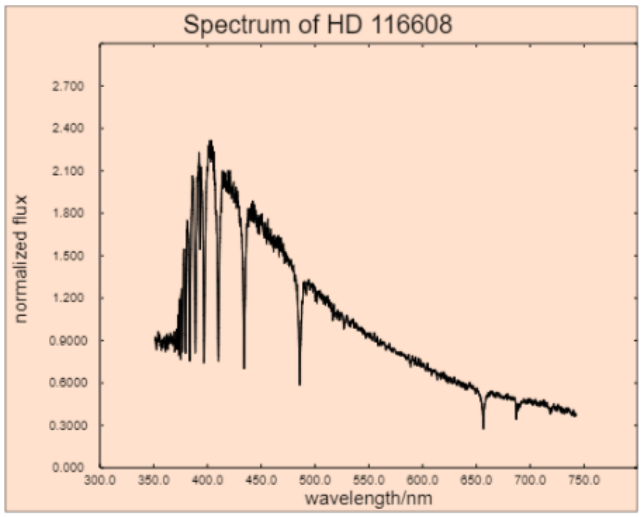
Wir wissen jetzt, wie der Stern heißt. Wenn Sie im Internet danach suchen, finden Sie eine Fülle von Informationen darüber. Damit wir den Ladevorgang mit anderen Daten wiederholen können, kapseln wir ihn in einem eigenen Block. Nach dessen Ausführung liegen die eigentlichen Sterndaten als leicht aufbereitete Tabelle vor. Unschön daran ist die stark unterschiedliche Größenordnung der Daten in den beiden Spalten. Wir normalisieren deshalb die zweite Spalte mithilfe des Mittelwerts und speichern das Ergebnis als ***normalized data***.

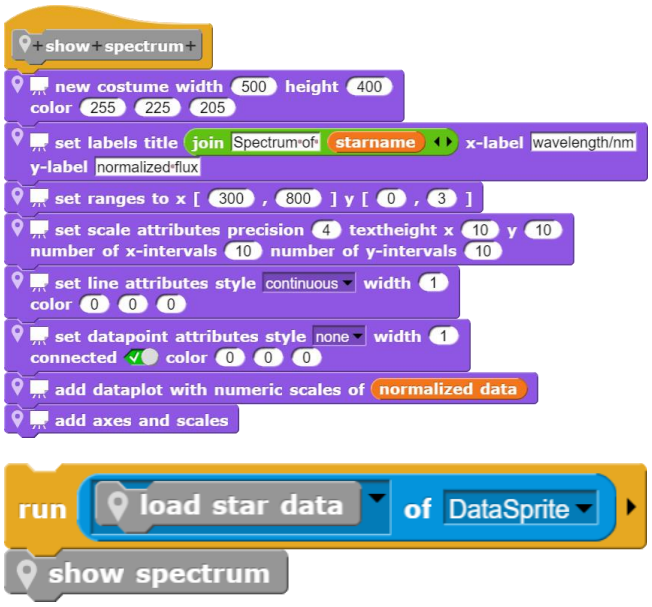




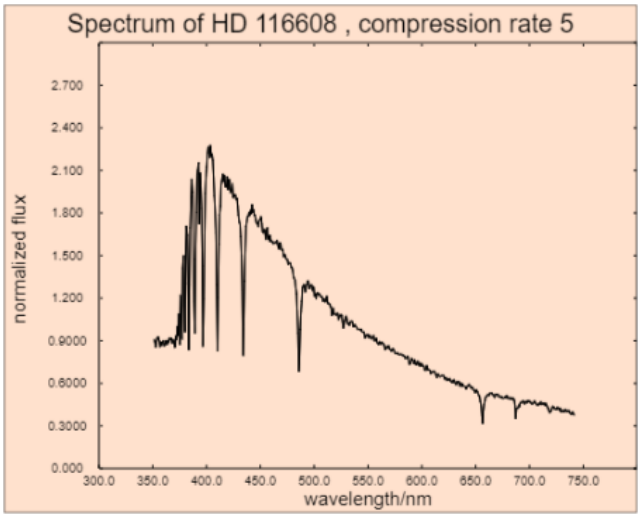
Damit hat das *DataSprite* erstmal seine Schuldigkeit getan. Wir wechseln zum *PlotSprite*.

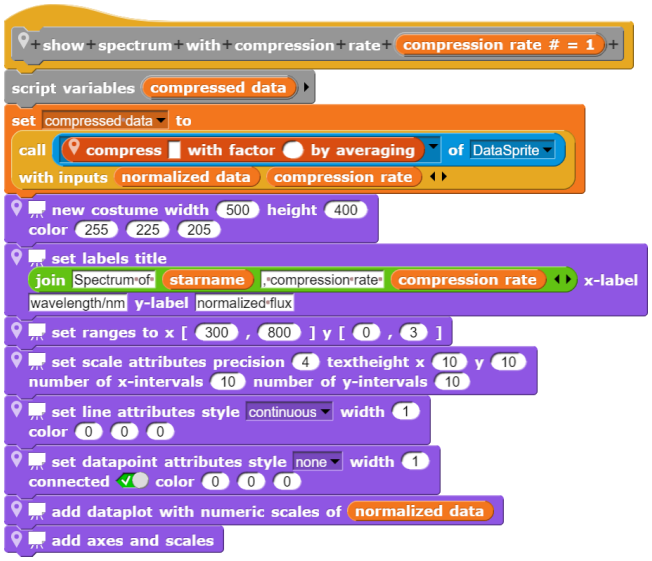
Mit den normalisierten Daten lässt sich schnell ein Diagramm im *PlotSprite* erstellen.



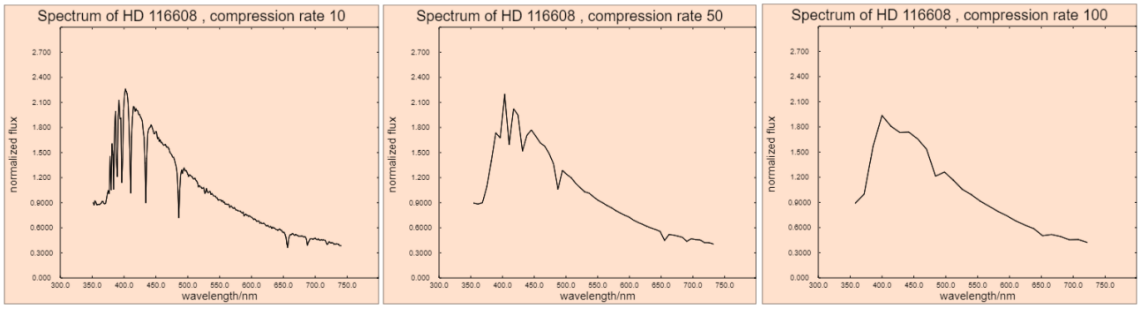


Man erkennt gut den abfallenden Verlauf mit einigen markanten Absorptionslinien. Aber braucht man für diese Erkenntnis überhaupt alle Spektraldaten? Vielleicht genügt es ja, durch Mittelwertbildung die Datenmenge zu reduzieren. Wir führen einen Kompressionsfaktor ***compression rate*** ein und ergänzen das Skript vor der Diagrammerstellung.



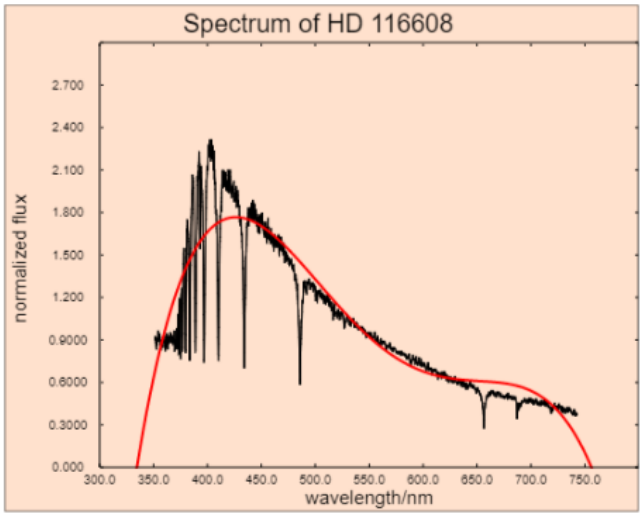


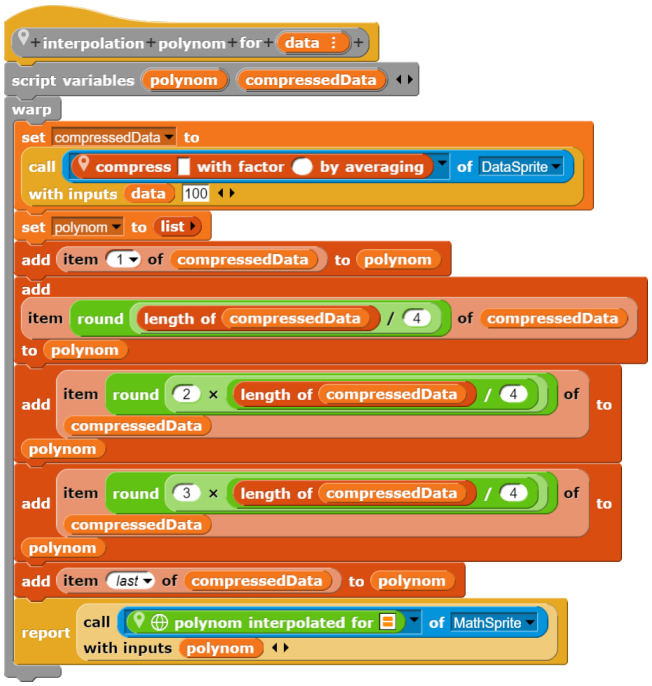
Der Faktor 5 ändert nicht viel.
Probieren wir also weiter.



Man sieht, dass der temperaturabhängige Verlauf des Spektrums kaum verändert wird.
Nur die Absorptionslinien gehen verloren.

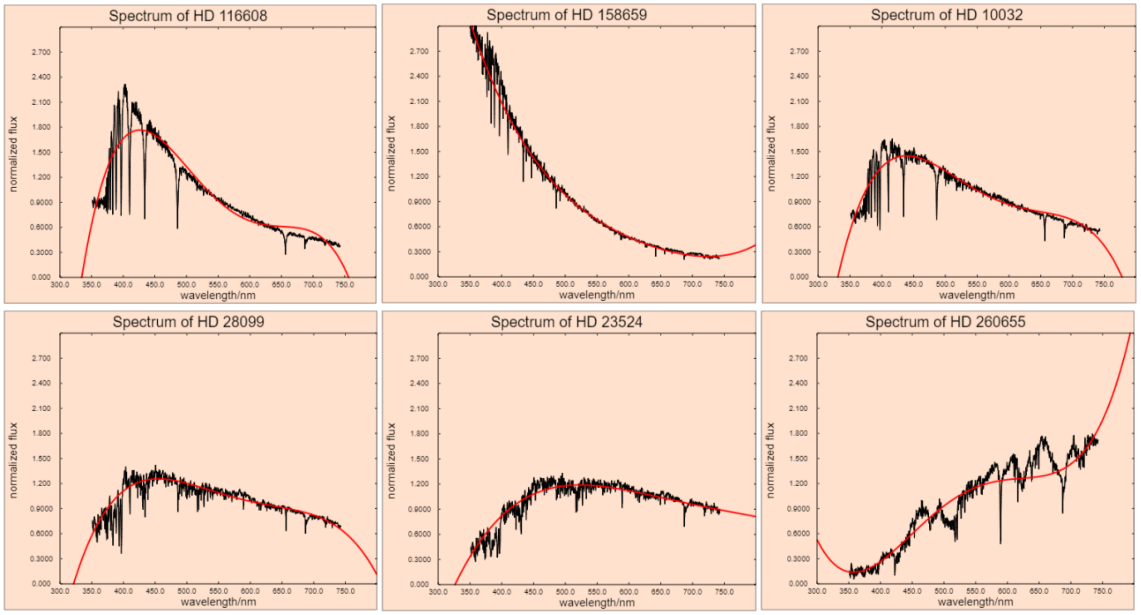
Somit sollte sich der Typ des Spektrums durch ein Interpolationspolynom z. B. 4. Grades beschreiben lassen. Wir laden dazu zusätzlich das *MathSprite* und probieren es so:





Das funktioniert also hervorragend! Protokollieren wir die Polynomparameter bei der Untersuchung gleich mit, dann können wir anhand der Parameterbereiche die Sterntypen leicht unterscheiden.

7	A	B	C	D	E	F
1	star name	a4	a3	a2	a1	a0
2	HD 116608	-1.2493580327340172e-9	0.000002886862108780081 4	-0.002459579857425176	0.9103088865090065	0.9103088865090065
3	HD 158659	1.5652590170171 66e-10	-3.9630320801781 07e-7	0.0003879661846290463	-0.17622312866994078	-0.17622312866994078
4	HD 10032	-7.27005023271818e-10	0.000001694929847991264	-0.001462425925103779	0.5500801501694278	0.5500801501694278
5	HD 28099	-4.0018935572381893e-10	9.399457129604694e-7	-0.0008209889485783107	0.3141072191721327	0.3141072191721327
6	HD 23524	-8.18301248511472e-11	2.3253458278204257e-7	-0.0002461580054487 6965	0.11348374829256708	0.11348374829256708
7	HD 260655	6.248027476637483e-10	-0.000001337322548726115	0.0010450333683869723	-0.3486709605339992	-0.3486709605339992

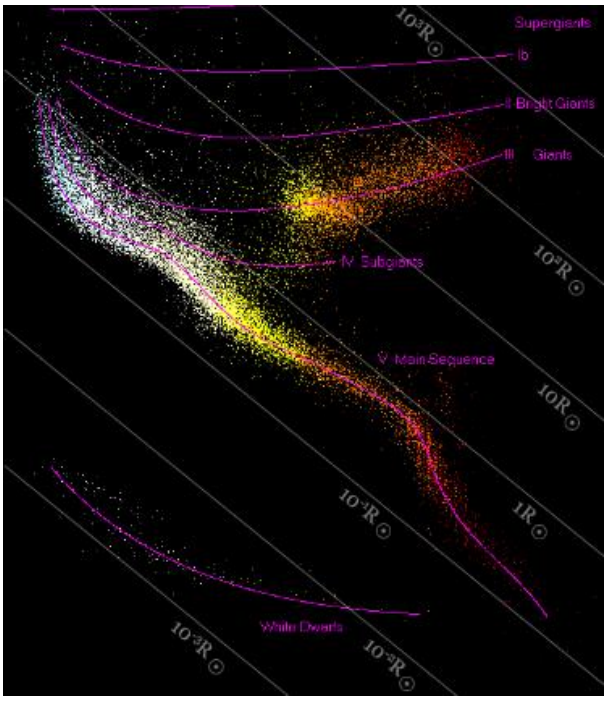


Füttert man mit den Polynomkoeffizienten ein Neuronales Netz, dann lernt dieses schnell, ein Diagramm grob einem Sterntyp zuzuordnen. Das Programm kann anhand der alten Daten also „lernen“ welche Parameterintervalle zu welchen Sternklassen gehören. Gibt man die Daten eines neuen Sterns ein, dann ermittelt es die Koeffizienten des Polynoms und gibt danach eine gut begründete Prognose ab, um welche Art von Stern es sich handeln könnte.

Aufgaben:

1. Stellen Sie für die nicht komprimierten Spektrumsdaten jeweils ein Interpolationspolynom möglichst niedrigen Grades auf. Welche Punkte sollten dafür gewählt werden? Ergeben sich Unterschiede zwischen diesen Polynomen und den Ergebnissen des oben gezeigten Verfahrens?
2. Entwickeln Sie ein Skript, das ein unbekanntes Spektrum einem der bisher auftretenden Typen zuordnet.
3. Entwickeln Sie ein Verfahren, um die markantesten Absorptionslinien genauer zu untersuchen. Stellen Sie diese für Sterne der gleichen Klasse vergrößert dar und versuchen Sie, Unterschiede „automatisch“ zu bestimmen. Diskutieren Sie Ihre Ideen vor der Realisierung.

5.4 Klassifizierung von Sternen nach dem kNN-Verfahren

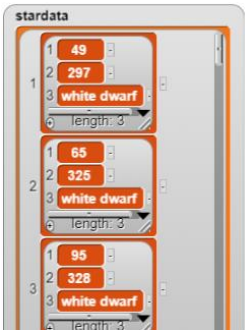


Im Hertzsprung-Russel-Diagramm (s. Wikipedia) wird die Leuchtkraft von Sternen über ihrer Sternklasse aufgetragen. Es ergibt sich einen Art Linie von links-oben-nach rechts-unten, die „Hauptreihe“. Auf dieser halten sich Sterne wie die Sonne überwiegend auf. Rechts-oben über der Hauptreihe finden wir die Roten Riesen, links-unten unter der Hauptreihe die Weißen Zwerge. Das recht erstmal. (Bildquelle: [HR])

Wir wollen neue Sterne in diesem Diagramm klassifizieren, indem wir das Verfahren der k-nächsten-Nachbarn (kNN) verwenden: Wir erzeugen als Trainingsdaten eine Liste von Sternen mit deren Koordinaten (einfach als Bildkoordinaten im Diagramm) und deren Typ. Wollen wir einen neuen Stern klassifizieren, dann bestimmen wir seine Position im Diagramm und suchen die nächsten k (z. B. k=5) Nachbarn. Danach bestimmen wir den am häufigsten auftauchenden Sterntyp in dieser Liste. Den weisen wir dem neuen Stern zu.

Zuerst einmal benötigen wir ein Bild des Hertzsprung-Russel-Diagramms ([HR]). Dieses importieren wir in *Snap!* als Kostüm eines *ImageSprites* und erzeugen daraus die benötigten Daten.

Die Trainingsdaten erzeugen wir, indem wir einen Sterntyp vorgeben und danach einige Punkte im Diagramm anklicken, die diesem Typ entsprechen.

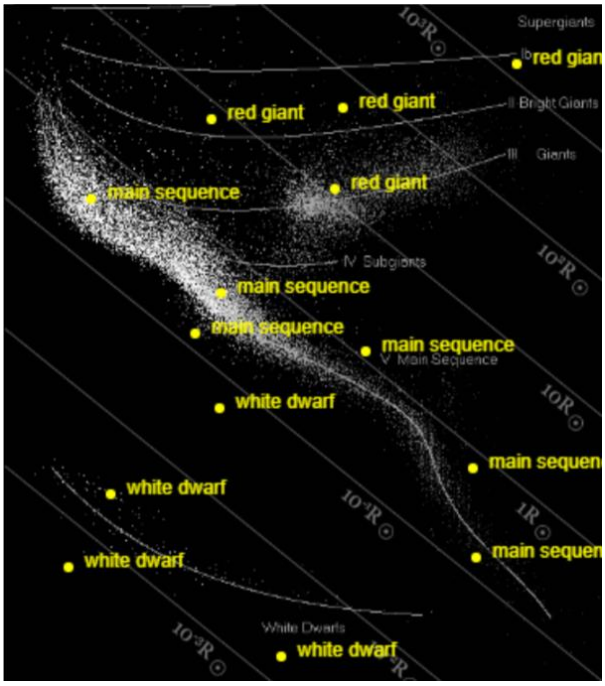


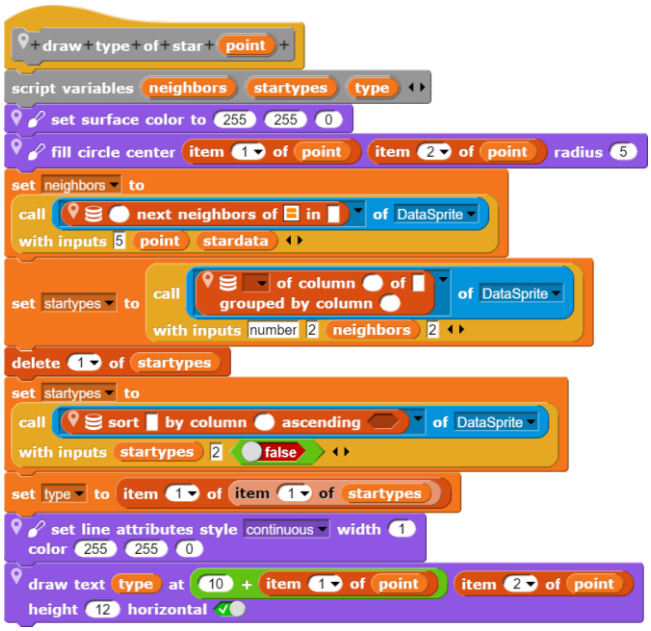
Da wir auf dem Bild zeichnen wollen, arbeiten wir mit einer Kopie des HR-Diagramms, um das Original nicht zu verändern.

Danach können wir neue Sterne klassifizieren, indem wir sie anklicken und beschriften.

Wir stellen einige Properties für die Darstellung ein, …
… und zeichnen einen Kreis am Ort des Sterns.
Danach bestimmen wir die fünf nächsten Nachbarn und die Anzahlen des Auftretens ihres Typs. Im Ergebnis löschen wir die Überschriften und sortieren die Liste absteigend. Der Typ des neuen Sterns steht dann als erstes Element in der ersten Zeile. Diesen schreiben wir neben den Stern.

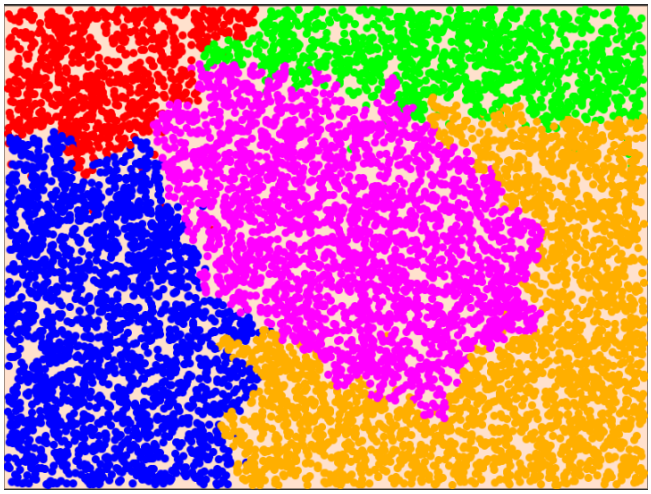
Das Ergebnis:





Aufgabe:

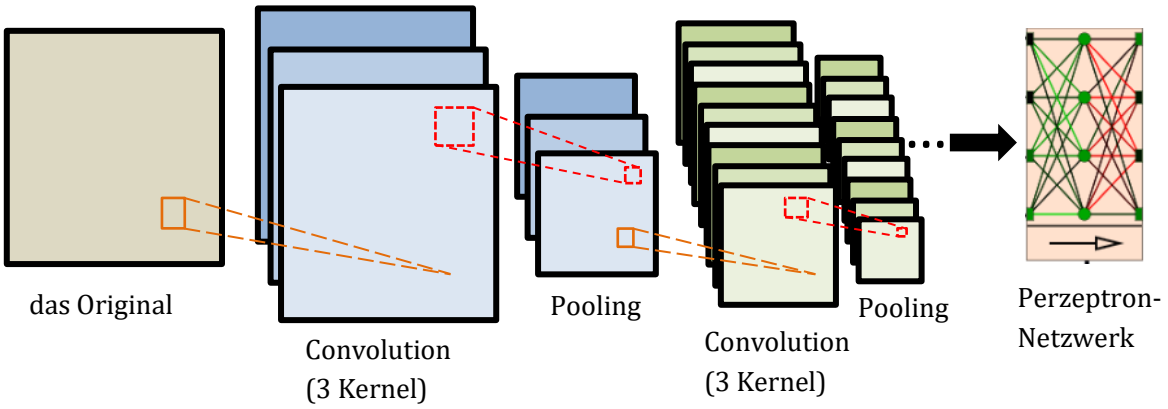
1. Fügen Sie die neu klassifizierten Sterne der Beispielliste hinzu, sodass sie bei weiteren Klassifizierungen mit hinzugezogen werden.
2. Zeichnen Sie für die unterschiedlichen Sternarten unterschiedlich farbige Punkte an den richtigen Stellen auf das Sprite, statt sie zu beschriften.
3. Lassen Sie den Prozess für zufällig ausgewählte Punkte ablaufen. Entsteht immer das gleiche Muster? Entstehen völlig verschiedene oder ähnliche Muster? Wovon hängt das ab?



5.5 Zeichenerkennung mit einem Convolutional Network

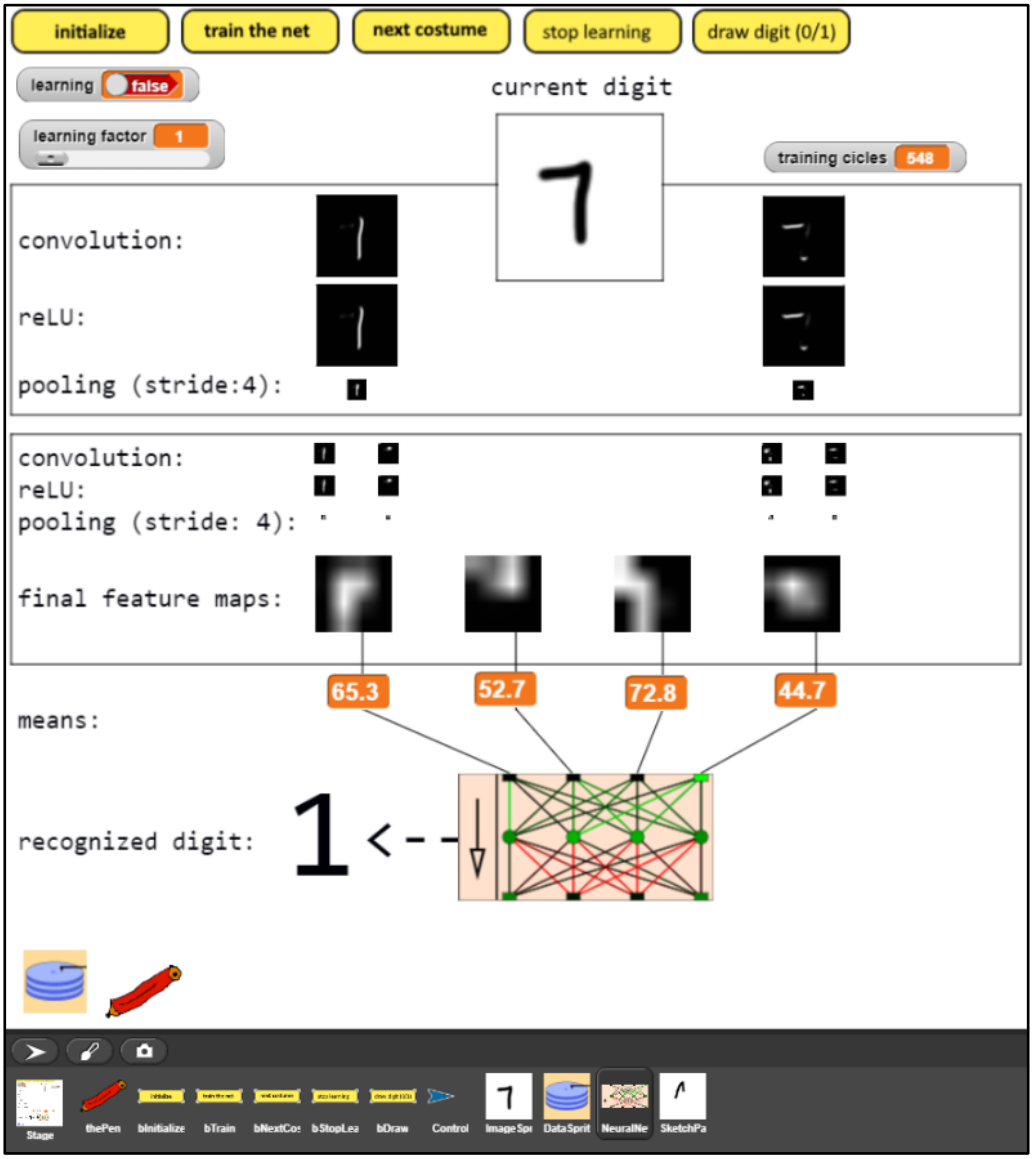
Die immense Zahl von Parametern in vollständig verbundenen Perzeptron-Netzen und der daraus folgende Bedarf an riesigen Mengen von Trainingsdaten hat zu anderen Netz-Varianten geführt, um diese Zahl drastisch zu reduzieren. Eine davon sind die *Convolutional Neural Networks (CNNs)*, bei denen die Eingabedatenmenge für das Perzeptron-Netz reduziert wird. Diese Art von Netzen wird z. B. in der Bild- und Spracherkennung sehr erfolgreich eingesetzt.

CNNs reduzieren die Datenmenge, indem in einem mehrstufigen Prozess zuerst mehrere Kernels angewandt werden, die bestimmte Eigenschaften z. B. eines Bildes (Kanten, ovale Flächen, …) herausfiltern und so zu mehreren *Feature-Maps* führen, die üblicherweise die gleiche Größe haben wie das Original. Das vergrößert erstmal die Datenmenge. Anschließend wird meist eine nichtlineare Aktivierungs-Funktion (*reLU*) auf die Feature-Maps angewandt und anschließend eine *Pooling*-Operation, die die Datenmenge wieder verkleinert. Meist handelt es sich um *Max-Pooling*, bei dem der Maximalwert aus einem Ausschnitt der Daten bestimmt wird. Macht man das mit einem „Fenster“, das mit einer bestimmten Schrittweite (*stride*) über die Feature-Map bewegt wird, dann erzeugt jeder Pooling-Schritt einen Wert der nächsten, verkleinerten Feature-Map.



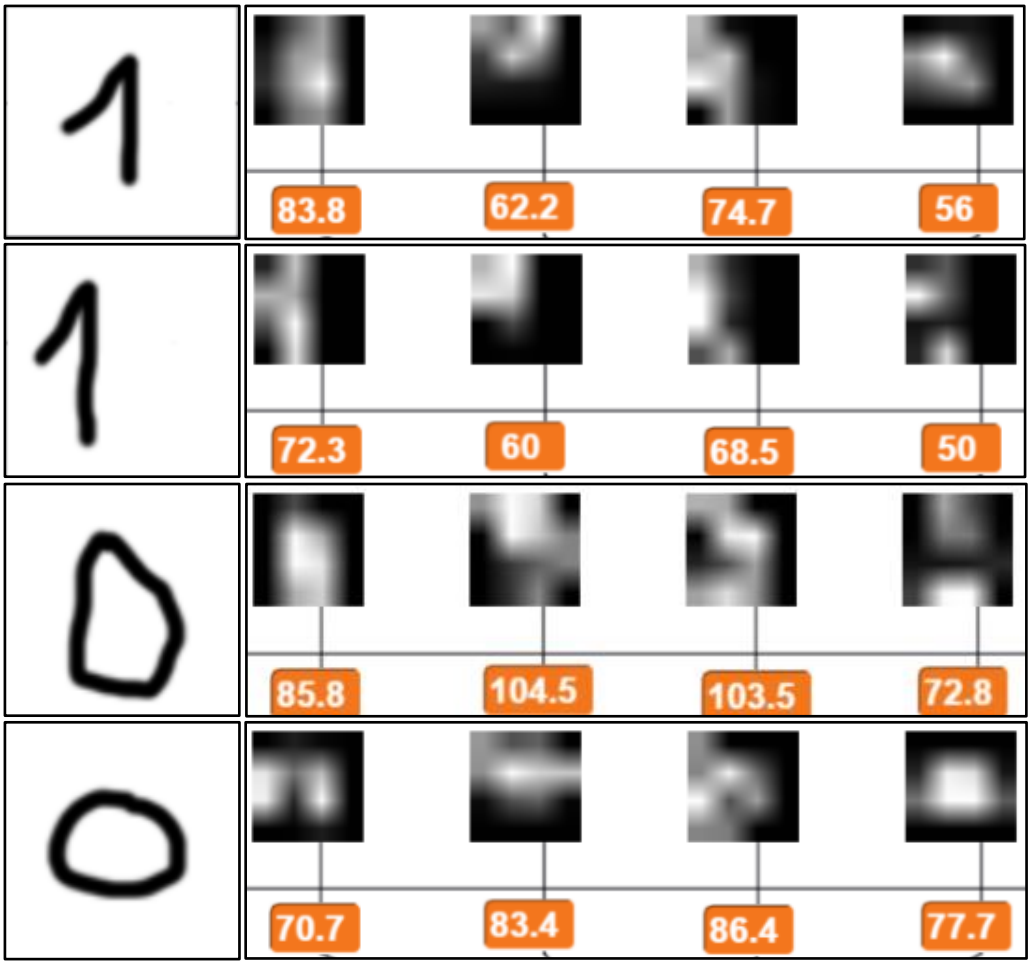
Als Beispiel nehmen wir einen Kernel, der senkrechte Linien filtert: er färbt einen Punkt weiß, wenn sich neben dem Punkt ein zweiter Bildpunkt befindet, sonst schwarz. Im „gefalteten“ Bild können wir dann senkrechte Linien des Originals als helle Flecken erkennen. Kommt es nicht so sehr darauf an, wo genau diese Linien sind, dann verlieren wir nicht allzu viel Information beim Pooling. Ein weißer Punkt in einer Feature-Map nach diversen Pooling-Prozessen bedeutet dann: *„In diesem Bereich befand sich irgendwo eine senkrechte Linie.“* Anhand solcher Daten aus mehreren Feature-Maps kann dann z. B. abgeleitet werden, dass sich dort auch eine horizontale Line, also eine Ecke befand. Hätten wir nach „beigen“ Bereichen sowie „ovalen“ Formen gesucht, dann wäre die Chance, Gesichter zu identifizieren, gar nicht so schlecht.

Wir wollen jetzt ein Modell für so ein CNN bauen, das die handgeschriebenen Ziffern *Null* und *Eins* unterscheiden kann. Dazu benutzen wir ein *DataSprite* für Hilfsoperationen, ein *ImageSprite* für das eigentliche Bild und – natürlich – ein *NeuralNetSprite* für das Perzeptron-Netzwerk am Ende der Kette. Ein weiteres, „normales“ Sprite namens *Control* soll die Abläufe steuern. Damit das Modell leichter zu bedienen ist, ergänzen wir es um einige Buttons sowie einen Stift, um die Oberfläche übersichtlich zu gestalten. Im Screenshot befindet sich das zu analysierende Bild oben im Kästchen, das Neuronale Netz zeigt sein Ergebnis unten an. Dazwischen werden von oben nach unten die verschiedenen Zwischenschichten durchlaufen und angezeigt. Als Zugabe enthält das Modell noch die Möglichkeit, eigene Ziffern zu zeichnen.



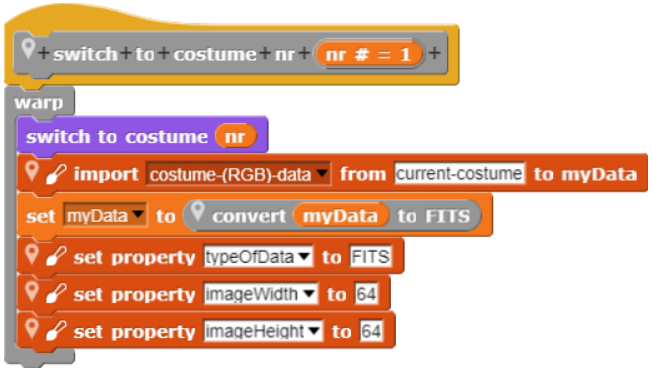
Unser CNN wird mit je 10 Ziffern aus je 64x64 Pixeln für die Nullen und Einsen trainiert. Anschließend soll es diese sowie andere handgeschriebene „erkennen“. Eigentlich müssten wir mehrere Kernel unseres CNN speziell für diese Aufgabe trainieren. Stattdessen nehmen wir nur zwei bekannte Kernel zur Erkennung vertikaler und horizontaler Linien, weil sich durch die Beschränkung auf zwei alles am Bildschirm darstellen lässt und die Ergebnisse sogar halbwegs zu interpretieren sind. (Die Erkennungsrate leidet darunter allerdings heftig!) Trainiert wird also nur das Perzeptron-Netz mit vier Eingangswerten.

Im obigen Bild sind nach zwei Stufen der Reduktion vier Feature-Maps von je 16x16 Pixeln übrig, bei denen jeweils zweimal die Operationen *Convolution → reLU → Max-Pooling* durchlaufen wurden: ganz links mit dem Kernel für senkrechte Linien, dann mit beiden Kernels in unterschiedlicher Reihenfolge und zuletzt zweimal mit dem Kernel für horizontale Linien. Die Ziffern darunter geben den Mittelwert der Helligkeit gemessen über das gesamte Bild an. Wenden wir dieses auf verschiedene Ziffern an, dann zeigt sich die Möglichkeit, trotz des sehr ein fachen Verfahrens Unterschiede zwischen Nullen und Einsen zu messen.

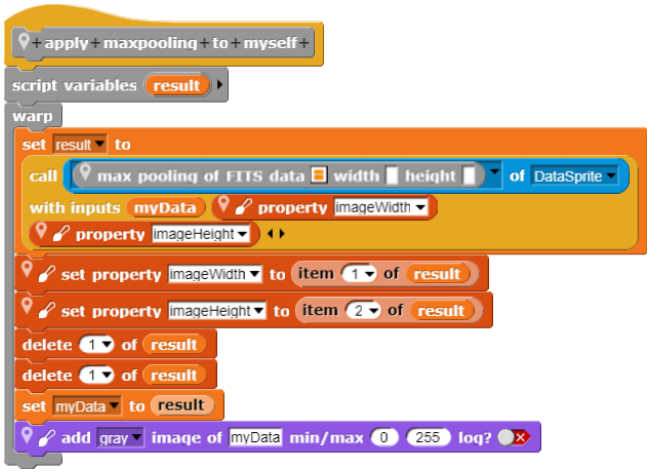


Betrachten wir die Funktionalitäten der einzelnen Objekte:

Das *ImageSprite* soll die Daten eines neuen Kostüms als Grauwerte in seinen Datenbereich importieren. Das geht sehr einfach.



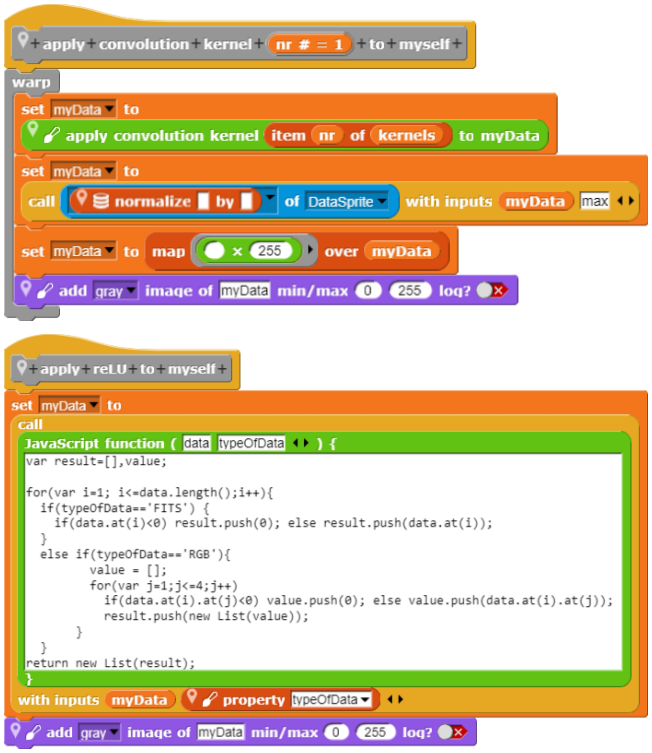
Weiterhin sollen das Sprite und seine Klone die drei Operationen eines CNN ausführen können. Mit etwas Hilfe des *DataSprites* geht auch das.

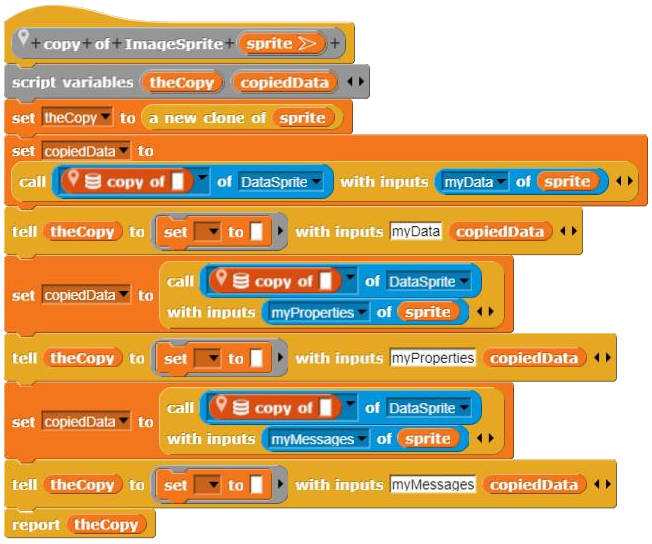


Mehr muss das *ImageSprite* für unsere Zwecke nicht zusätzlich beherrschen.

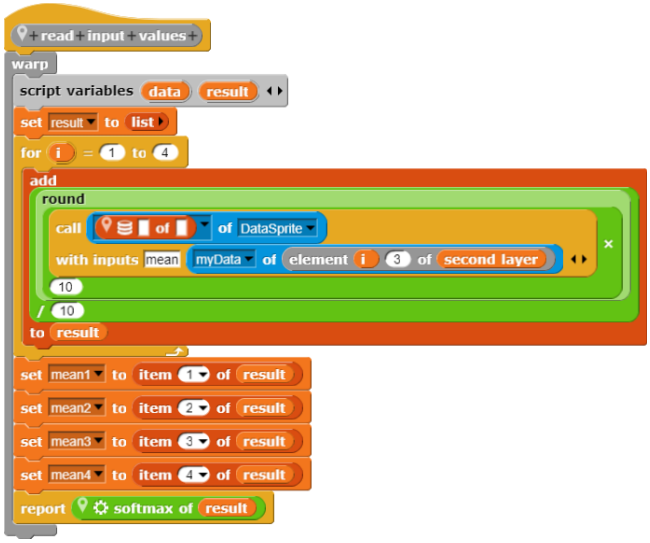
Das *Control*-Sprite muss das *ImageSprite* bitten, das Kostüm zu wechseln und dieses danach zu analysieren. Dabei hält es sich streng an die Vorgaben für CNNs. Die Methode ***init*** sorgt nur für das Zeichnen der Linien auf der Bühne. Die weiteren Methoden arbeiten mit zwei Ebenen des CNN, ***first layer*** und ***second layer***, die jeweils die auf der Bühne erscheinenden Versionen der Ziffern enthalten. Damit sich die nicht gegenseitig stören, wird mit Kopien des *ImageSprites* gearbeitet, nicht mit Klonen. Beim Kopieren hilft wieder das *DataSprite*.

Nachdem die erforderlichen Kopien erzeugt wurden, werden diese von Control gebeten, die jeweilige CNN-Operation auszuführen. Zuletzt werden die inzwischen ziemlich mickrigen (4x4 Pixel) Klone der letzten Ebene als „final feature maps“ stark vergrößert dargestellt. Mit diesen wird dann das Neuronale Netz trainiert.

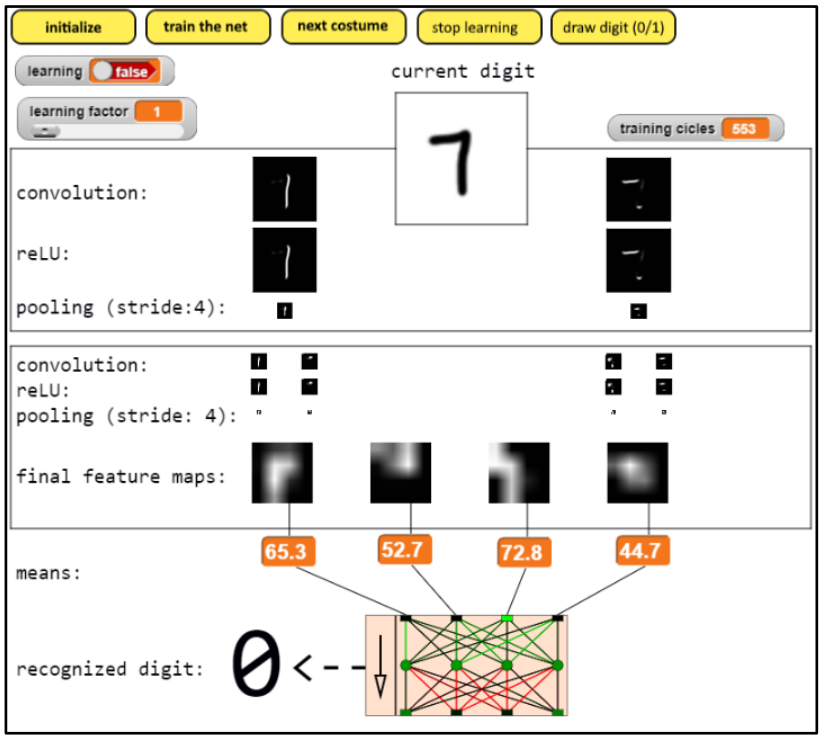




Das Neuronale Netz in Form eines *NeuralNetSprites* soll bei Nullen die größte Ausgabe am Ausgang 1, bei Einsen am Ausgang 4 erzeugen. Das ist natürlich völlig willkürlich. Den aktuellen Ausgabewert ermittelt die Funktion ***read output***. Mit dessen Komponenten kann das Netz trainiert werden, wenn es uns gelingt, aus der letzten Ebene von ***second layer*** die Mittelwerte zu bestimmen. Diese modellieren wir noch passend mit der ***softmax***-Funktion.



Und – hat das Netz was gelernt?



Nun ja – es gibt noch Verbesserungsmöglichkeiten!

Hinweise

Maschinelles Lernen besteht zu einem großen Teil aus der Aufbereitung von Daten – egal, ob es sich um Tabellendaten oder Bilder handelt. Die eigentlichen Lernvorgänge der Maschinen bestehen dann aus den Parameteranpassungen, die sich aus den Daten ergeben. Da beides gut visualisierbar ist, bietet sich ein breites Feld für Programmieranfänger mit vielen Übergängen zum Bereich „Informatik und Gesellschaft“.

Beispiele für die Anwendung der Operationen der *ML.SpriteLibrary*, insbesondere der Faltungen mithilfe eines Kernels, findet man reichlich in [DBV].

Liste der Beispiele

Literaturhinweise und Quellen

[Albon] Albon, Chris: Machine Learning Kochbuch, O'Reilly, 2919

[Baumann] Baumann, Rüdeger: Didaktik der Informatik, Klett, 1990

[Census] https://archive.ics.uci.edu/ml/datasets/census+income

[DBV] Burger, W., Burge, M.-J-: Digitale Bildverarbeitung – Eine Einführung mit Java und ImageJ, Springer 2006

[FITS] de.wikipedia.org/wiki/Flexible_Image_Transport_System

[Goodfellow] Goodfellow, I.; Bengio, Y.; Courville, A.: Deep Learning, MIT Press, 2016

[Grus] Grus, Joel: Einführung in Data Science, O'Reilly, 2016

[HOU] Hands-On Universe: handsonuniverse.org/

[HR] https://studylibde.com/doc/2985884/hertzsprung-russell--und-farb-helligkeits

[JSON] Popular Baby Names: https://catalog.data.gov/dataset/most-popular-baby-names-by-sex-and-mothers-ethnic-group-new-york-city-8c742

[NYcitibike] https://www.citibikenyc.com/system-data

[Minsky] Minsky, Marvin: Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, New York, 1967

[Modrow1] Modrow, Eckart: Technische Informatik mit Delphi, emu-online, 2004

[Modrow2] Modrow, Eckart: Zur Didaktik des Informatikunterrichts – Band 2, Dümmler, 1992

[Rojas] Rojas, Rau´l: Neural Networks - A Systematic Introduction, Springer Berlin, 1996

[SAP] www.sap.com/germany/products/leonardo/machine-learning.html

[SchulAstro] www.schul-astronomie.de

[SQL] Modrow, Eckart: Informatik mit Snap!, http://ddi-mod.uni-goettingen.de/InformatikMitSnap.pdf

[SZ] Süddeutsche Zeitung: 10. April 2019 www.sueddeutsche.de/wissen/schwarzes-loch-bild-1.4404130

[UniGOE] Institut für Astrophysik, Universitaet Goettingen