
CHALMERS | 
GÖTEBORGS UNIVERSITET

Funktionell PCA mot Artificiella Neuronnät

Jämförelse av metoder för bildigenkänning

Functional PCA vs Artificial Neural Networks

A comparison of methods for image recognition

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Kandidatarbete inom civilingenjörsutbildningen vid Chalmers

Erik Jansson
Freja Nordh
Jack Sandberg
Mattis Hallberg
Philip Gard
Shayan Mollahosseini

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2021

Funktionell PCA mot Artificiella Neuronnät

Jämförelse av metoder för bildigenkänning

Examensarbete för kandidatexamen i matematisk statistik vid Göteborgs universitet
Shayan Mollahosseini Mattis Hallberg Freja Nordh

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk fysik vid Chalmers
Erik Jansson Philip Gard

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers
Jack Sandberg

Handledare: Krzysztof Podgórski

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2021

Förord

Det här är ett kandidatarbete av jämförelse mellan funktionell principalkomponentanalys och artificiella neuronnät för bildigenkänning. Arbetet har utförts på distans av sex studenter från Chalmers tekniska högskola och Göteborgs universitet under våren 2021. Vi vill gärna tacka vår handledare Krzysztof Podgórski för sitt entusiastiska engagemang och sin vägledning under arbetets gång. Vi vill dessutom tacka alla föreläsare på Chalmers tekniska högskolan och Göteborgs universitet för sina intressanta och givande föreläsningar under de senaste tre åren som gett oss fördjupade kunskaper inom matematik (och fysik!).

En gemensam dagbok och individuell tidslogg har förts över samtliga skribenters bidrag. Alla har varit delaktiga under rapportskrivandet, detaljerade bidrag redovisas i tabellen nedan. Dock vill vi betona att hela rapporten är skriven gemensamt. Freja har varit ansvarig för kommunikation med handledare.

Veckovisa möten har hållits under projektets gång. Under mötena har arbetsuppgifter delats ut för kommande veckan/-orna. Vi har även diskuterat veckans problem, svårigheter vad gäller både teori, implementation, analys och skrivande. Både teori och klassificeringsmetoderna som utforskats i detta arbete har omfattat mer än vad som har varit möjligt (och relevant) att inkludera i rapporten. Exempelvis lades mycket tid på teorin för splines samt Karhunen-Loéves sats som senare bedömdes inte vara relevant nog för rapporten.

Avsnitt	**Huvudsaklig författare**	**Övrig information**
Populärvetenskaplig Sammanfattning & Abstract	Shayan Mattis	Reviderat av Freja
1 Inledning	Jack	Omskrivet av Jack. Erik, Shayan, Freja och Mattis har bidragit till tidigare versioner.
1.1	Mattis, Erik	
1.2	Shayan, Erik	Reviderat av Mattis
2 Teori	Erik	
2.1	Erik	
2.1.1	Erik	Reviderat av Philip
2.1.2	Erik	Nedkortat av Jack.
2.1.3	Jack	
2.2	Jack	
2.2.1	Jack	
2.3	Freja, Shayan, Mattis	
2.3.1	Freja, Shayan, Mattis, Jack	
2.3.2	Freja, Shayan, Mattis	
2.3.3	Freja, Shayan, Mattis, Jack	
2.4	Jack	
3 Dataset	Jack	
3.1	Jack	
3.1.1	Jack	
3.1.2	Jack	
3.1.3	Jack	
3.2	Erik	
3.3	Erik	
4 Metod	Jack	
4.1	Jack, Mattis	
4.2	Erik	
4.2.1	Erik	
4.2.2	Erik	

4.2.3	Jack	
4.3	Shayan, Mattis, Jack.	
5 Resultat		
5.1	Mattis	
5.2	Mattis	
5.3	Mattis	
5.4	Mattis	
6 Diskussion		
6.1	Erik, Freja	Reviderat av Mattis
6.2	Erik	
6.3	Erik	
6.4	Philip	
6.5	Philip	
6.6	Philip	
6.7	Erik	
6.7.1	Erik	
6.7.2	Erik	
7 Slutsats	Erik	Reviderat av Mattis
Referenser	-	Granskat av Shayan
A Ordlista	Shayan, Jack.	
B Modeller för neurala nätverk	Mattis	
C Plots av egenvärden för de olika klasserna	Erik	
D Bilden av första principalkomponenten för respektive klass	Erik	
E Kod	Jack	Mattis har bidragit med E.8.6-E.8.7, Erik har bidragit att ladda upp data till Colab för vissa körningar.
Figurer		
1	Erik, Jack	
2	Jack	
3	Shayan	
4	Shayan	
5	Jack, Mattis, Shayan	
6	Shayan, Mattis	
7	Jack	
8	Jack	
9	Erik	
10	Erik, Jack	
11	Erik, Jack	
12	Erik, Jack	
Tabeller		
1	Jack	
2	Erik	
3	Jack	
4	Mattis,Jack	
5	-	Struktur av Jack, data inmatad av Mattis.

Populärvetenskaplig presentation

Maskininlärning är den vetenskapliga studien av statistiska modeller och algoritmer. Datorsystem använder algoritmerna för att kunna utnyttja slutsatser och mönster, som i sin tur utför uppgifter istället för att använda tydliga instruktioner. Som en del av artificiell intelligens skapar algoritmerna till maskininlärningen en matematisk modell baserad på given information eller historik för att förutsäga eller fatta beslut utan uppenbar planering.

Man använder förmodligen maskininlärning flera gånger om dagen, även utan att veta om det. Varje gång en person gör en internetsökning på *Bing* eller *Google* används maskininlärning. När sociala medier såsom *Instagram* eller *Facebook* känner till någons vänner i bilder är det en konkret typ av maskininlärning. När ens e-postprogram sorterar skräppost från icke-skräppost så är det också maskininlärning.

Att maskinlärningen bygger på både algoritmer och statistiska modeller är just det som gör det möjligt att utföra bildigenkänning utan att förutsättningarna är helt perfekta. För att kunna uppfatta hur bildigenkänning fungerar så måste man bryta ned den i sina beståndsdelar. Nyckeln till att kunna göra det är kännedom till Artificiella neuronnät och mer specifikt faltande neuronnät, CNN samt olika statistiska metoder såsom funktionell principalkomponentanalys, FPCA.

Bildigenkänning är en gren inom det tvärvetenskapliga området datorseende, som handlar om maskinlärning och hur algoritmer och statistiska modeller används för att identifiera och klassificera bilder. Bildigenkänning används inom olika områden, exempelvis för automatisk bildorganisation från molnapplikationer till telekommunikationsföretag, bildförsäljning på fotograferings- och filmswebbplatser, visuell sökning för att upptäcka produkter, bildklassificering för webbplatser med stora visuella databaser, bild- och ansiktsbehandling på sociala medier samt marknadsföring och kreativa kampanjer.

Detta arbete handlar inte enbart om maskininlärning och bildigenkänning utan om precision hos några olika metoder för bildigenkänning. För att kunna uppfatta ett av många användningsområden och tillämpningar till ett sådant arbete, är det väsentligt att ha kännedom om maskinlärning.

Datamängden som vi har använt innehåller totalt 70,000 unika bilder av klädesplagg från modeföretaget *Zalando*. De beaktade modellerna är gjorda för att kunna identifiera vad som visas på varje bild samt kunna klassificera dem. Bilderna laddades ner, reducerades i storlek till 28×28 pixlar samt omvandlades till gråskala. Sen delades bilderna upp i två olika grupper: en testgrupp och en träningsgrupp. Träningsgruppen användes för att träna en modell att klassificera bilderna. Testgruppen användes för att säkerställa att modellen klarar av att klassificera bilder som den inte tidigare har observerat. Det är en kontroll vi gör för att kunna avgöra om modellen kan generalisera träningsgruppen till testgruppen eller om den enbart memorerar träningsgruppen. Med metoden klassifiering med medelvärde så fick vi en noggrannhet på 68.6%. Liknande princip användes på ett antal andra metoder med samma träningsgrupp och testgrupp. Resultaten visar att metoden klassifiering med faltande neuronnät, CNN, har bäst noggrannhet på 90,4% bland de använda metoderna.

Sammanfattning

Denna rapport fokuserar på jämförelsen av några olika klassificeringsmetoder applicerade på bilddatan Fashion-MNIST. De olika metoderna är artificiella neurala nätverk och funktionell principalkomponentanalys och principalkomponentanalys. För de neurala nätverken har vi två typer: CNN och FNN. Den förstnämnda är specialiserad på just bilder medan den sistnämnda kan appliceras på olika typer av dataset fast har nackdelen med försämrad noggrannhet. Funktionell principalkomponentanalys eller FPCA är en utvidgning av principalkomponentanalys (PCA) som innebär studiet av dimensionreducering av högdimensionell data. FPCA uttrycker data i form av funktioner vilket möjliggör ytterligare dimensionsreducering om funktionerna effektivt representerar datan. Parametrar i majoriteten av metoderna bestäms med hjälp av korsvalidering. Korsvalidering tillämpas för att en modell inte ska bli partiskt mot en viss del av datan och vi kan på så vis kan dra rättvisa slutsatser. Resultaten ger att de neurala nätverken, speciellt CNN, är bäst med en noggrannhet över nittio procent. Däremot presterar FPCA och PCA också bra då de har en noggrannhet omkring åttio procent.

Nyckelord: Artificiella neuronnät, bildigenkänning, Fashion-MNIST, funktionell principalkomponentanalys, klassificering, korsvalidering, neurala faltningsnät, principalkomponentanalys, splines.

Abstract

This report focuses on the comparison of some classification methods on image data. The methods are artificial neural networks and functional principalcomponentanalysis. For the neural networks we use two types: CNN and FNN. The first is specialized on image data while the latter can be applied to different data sets but with the drawback of decreased precision. Functional principalcomponentanalysis or FPCA is an extension of principalcomponentanalysis (PCA) which is the study of dimension reduction of high dimensionell data. FPCA expresses data in form of functions wich enables further dimension reduction if the functions represent the data well. The parameters in the majority of the models is decided with cross-validation. Cross-validation is applied to avoid biased models and it make sure that our conclusions are not biased. Our results show that the neural networks, especially CNN, is best with a precision above ninety percent. However the precision for FPCA and PCA is also good since it is around eighty percent.

Keywords: Artificial neural networks, classification, convolutional neural networks, cross-validation, Fashion-MNIST, functional principal component analysis, image recognition, principal component analysis, splines.

Innehåll

1 Inledning

Ämnet bildigenkänning har tillämpningar inom många olika discipliner som medicinsk bildbehandling, analys av satellitbilder och organisering av bildsamlingar [1]. Fältets popularitet har ökat kraftigt under de senaste 10 åren då stora genombrott har förbättrat möjligheten att klassificera bilder. Ett av de större genombrotten kom 2012 då AlexNet [2] övertygande vann tävligen ImageNet Large Scale Visual Recognition Challenge 2012[3]. Tävligens klassificeringsdel gav de tävlande 10,000,000 märkta bilder från över 10,000 kategorier (klasser) som de skulle använda för att träna en modell som kan klassificera bilder till rätt klass.

AlexNet är ett artificiellt neuronnät av typen faltande neuronnät. Artificiella neuronnät är en modell av hur neuroner i biologiska hjärnor interagerar. De första modellerna av biologiska neuroner har sitt ursprung redan på 1940-talet då Warren McCulloch och Walter Pitts modellerade ett enkelt neuronnät med elektriska kretsar [4]. Komplexiteten i de artificiella neuronnäten har ökat enormt sedan dess och neuronnäten finns i många olika varianter. En av dessa varianter är faltande neuronnät vars styrka är att faltningen utnyttjar spatiala strukturer i bilder för att effektivt klassificera [5]. Exempel på spatiala strukturer i en bild är hur närliggande pixlar interagerar för att skapa lokala mönster som en linje.

Från ett funktionellt perspektiv kan bilder ses som kontinuerliga funktioner där pixlarna fås från att evaluera en funktion i diskreta punkter. Med det funktionella perspektivet tillkommer en mängd analysmetoder från fältet funktionell dataanalys. En vanlig analysmetod inom funktionell dataanalys är funktionell principalkomponentanalys. Med funktionell principalkomponentanalys bryts den funktionella datan upp i principalkomponenter som beskriver datans beteende [6]. Principalkomponenterna kan sedan användas för att projicera datan till ett rum med färre dimensioner samtidigt som datan förlorar minimal information. I detta lågdimensionella rum är förhoppningen att man underlättat modellers förmåga att klassificera den ursprungliga datan genom att undvika effekter så som *the curse of dimensionality* [7].

I detta arbete studeras dels funktionell principalkomponentanalys och dess applikation inom klassificering av bilder. Dessutom görs en kortare genomgång av artificiella neuronnät där specifikt typerna fullt anslutna neuronnät samt neurala faltningsnät är av intresse. Dessa typer av neuronnät är sen tidigare välundersökta och tillhör de mest populära metoderna att tillämpa inom maskininlärning [8]. Metodernas klassificeringsförmåga jämförs slutligen på en datamängd bestående av klädesplagg. Klassificeringsmetoderna baserade på principalkomponetanalys ges extra förklaringsutrymme jämfört med metoder baserade på neuronnät då de senare anses vara väl undersökta och förklarade i andra arbeten.

1.1 Syfte

Syftet med studien är att empiriskt jämföra noggrannheten hos olika metoder för bildigenkänning baserade på principalkomponentanalys, funktionell principalkomponentanalys och neuronnät. För att jämföra dessa metoder krävs en teoretisk förståelse för metoderna samt implementation av metoderna i kod. Därför är två delsyften med arbetet att först få en uppfattning av den bakomliggande teorin, där principalkomponentanalysen ges extra fokus, och vidare implementera algoritmerna i kod för att mäta noggrannheten.

1.2 Avgränsningar

Datan som har använts har varit svartvita bilder av tio olika klädesplagg, med 28^2 pixlar. Resultaten som presenteras blir därför specifika för detta dataset och det förblir okänt hur de olika metoderna presterar på andra typer av bilder, av till exempel andra föremål eller bilder med högre upplösning.

Ytterligare kan bildigenkännings-metoder jämföras på många olika sätt, exempelvis beräkningstiden och hur mycket minneskapacitet som krävs. Dessutom kan man undersöka hur de olika metoderna skalar i beräkningstid eller minneskapacitet med ökande storlek på datan. På grund av arbetets omfång har vi främst fokuserat på att jämföra metodernas noggrannhet och simplicitet.

Slutligen har vi också använt oss av färdiga paket vid implementeringen av de olika metoderna. Vid funktionell principalkomponentanalys användes R-paketet `Splinets` [9] och för neuronnät användes maskininlärningspaketet *Keras* [10] via *Google Colab* [11]. Deras utformning och algoritmer har inte undersökts på djupet då vårt fokus har varit att använda dessa paket för att implementera våra metoder.

2 Teori

Denna teoridel inleds med en repetition om vektorrum, ortogonal projektion och baser. Därefter förklaras funktionerna som kallas *splines* i avsnitt 2.1.2 och senare hur data kan projiceras på splines med hjälp av R-paketet `Splinets` i sektion 2.1.3. Vidare är del 2.2 en särskilt viktig sektion som tar upp begreppen principalkomponenentanalys och funktionell principalkomponenentanalys, som är viktiga för att senare förstå metod-avsnittet 4. Därefter följer en genomgång av artificiella neuronnät under sektion 2.3, som behandlar bakgrund och implementering. Teorin avslutas med sektion 2.4 med att förklara begreppet korsvalidering.

2.1 Vektorrum och funktionsrum

Ett *vektorrum*[12] är en icke-tom mängd V med en additionsoperator mellan element i mängden samt där elementen kan multipliceras med skalärer. Ett vektorrum måste uppfylla 10 krav, bland annat ska V vara slutet under addition och det ska finnas ett 0-element. Några exempel på vektorrum är $\mathbb{R}^n$ för geometriska vektorer, samt $F(I)$ som är mängden av alla skalärvärda funktioner på ett intervall I. Elementen i vektorrummen kan alltså vara vektorer, men kan likväl vara funktioner. Vi kallar en delmängd av $F(I)$ för ett *funktionsrum*, exempelvis $C[a,b]$, mängden av alla kontinuerliga funktioner på $[a,b]$.

En viktig typ av vektorrum är *Hilbertrum*[13]. Hilbertrummet $\mathcal{H}$ är ett vektorrum med en inre produkt som är fullständigt med avseende på den inre produkten. Att $\mathcal{H}$ är fullständigt betyder att varje Cauchyföljd som tillhör $\mathcal{H}$ konvergerar mot en gräns i $\mathcal{H}$. Vidare är ett särskilt viktigt Hilbertrum det så kallade $L^2[a,b]$-rummet, som består av alla funktioner $f : (a,b) \to \mathbb{C}$ med den inre produkten

$$\langle f,g\rangle = \int_a^b f(t)\overline{g(t)}dt \tag{1}$$

som dessutom är Lebesgue-mätbara och kvadratiskt integrerbara. Slutligen kallas $||f||_2 = \sqrt{\langle f,f\rangle}$ för L_2-normen av funktionen f.

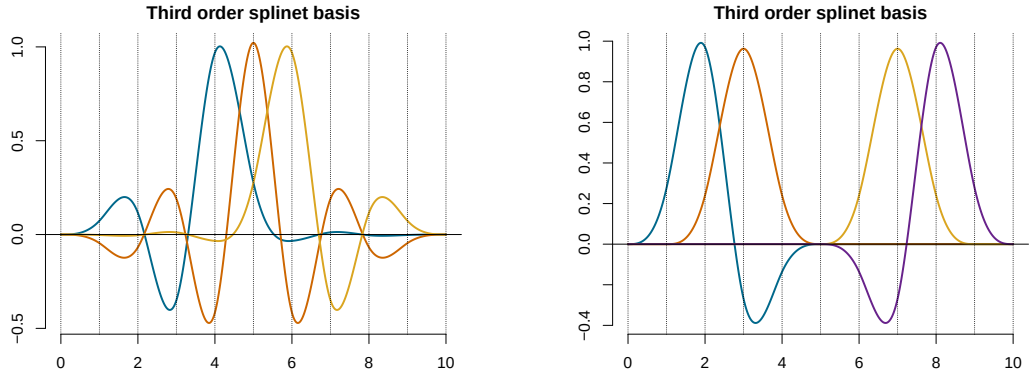
2.1.1 Ortogonal projektion och baser

I denna del förklaras först begreppet *ortogonal projektion*[14] och senare begreppet *bas*[12].

Om vi låter $\mathcal{M}$ vara ett slutet underrum av Hilbertrummet $\mathcal{H}$ och $h \in \mathcal{H}$, då kan man visa att det finns ett unikt element $m_0 \in \mathcal{M}$ sådant att $h - m_0 \in \mathcal{M}^\perp$, där $\mathcal{M}^\perp$ är det *ortogornala komplementet* till $\mathcal{M}$. Då finns en operator $P : \mathcal{H} \to \mathcal{M}$ som kallas den *ortogonala projektionen* av $\mathcal{H}$ på $\mathcal{M}$ som definieras av $Ph = m_0$ och uppfyller:

- P är en linjär avbildning på $\mathcal{H}$
- $\|Ph\| \leq \|h\| \ \ \forall h \in \mathcal{H}$
- $PPh = Ph$

En *bas* för $\mathcal{H}$ är en uppsättning element $h_1, ..., h_n \in \mathcal{H}$ sådana att de spänner upp $\mathcal{H}$ och är linjärt oberoende. Precis som att en bas av vektorer kan spänna upp geometriska rum kan begreppet bas även tillämpas för funktionsrum. Principen är densamma, men lite mer abstrakt för fallet med funktionsrum. Basen kan även ha egenskaper som ortogonal och normerad. Att två funktioner är ortogonala innebär samma sak som för vanliga vektorer, att deras inre produkt är 0.



Figur 1: Tredje ordningens splinet-bas över noderna [0,1,2,...,10].

2.1.2 Splines

Splines[15] är kontinuerliga styckvisa polynom med given ordning k som är kopplade genom ett antal noder $[\xi_0, ...,\xi_{n+1},]$. I noderna kan en spline ha kontinuerlig derivata av högst ordning $k-1$. Domänen för en spline kallas för dess omfång och antas vara ett ändligt intervall. Den första och sista noden kallas för ändnoder och övriga kallas för inre noder.

En splines utseende mellan två noder, $[\xi_n, \xi_{n+1}]$ beror på polynomen i de omkringliggande intervallen $[\xi_{n-1}, \xi_n]$ och $[\xi_{n+1}, \xi_{n+2}]$ eftersom splines har ett kontinuietskrav i noderna. På grund av detta påverkar ändnoderna splinen annorlunda jämfört med de inre noderna vilket leder till att ändnoderna brukar få speciella krav. Det visar sig vara smidigt att låta splinen och alla dess derivator vara 0 i ändpunkterna förutom derivatan av högsta ordningen.

För en given mängd noder och ordning är rummet av splines ett ändlig-dimensionellt underrum av Hilbertrummet $L^2[a,b]$. Alltså kan en godtycklig funktion i $L^2[a,b]$ projiceras på rummet för splines. Med en ortogonal bas kan funktionen sedan analyseras genom att betrakta projektionens olika baselement med dess koefficienter.

2.1.3 Omvandling av data till spline via paketet Splinets

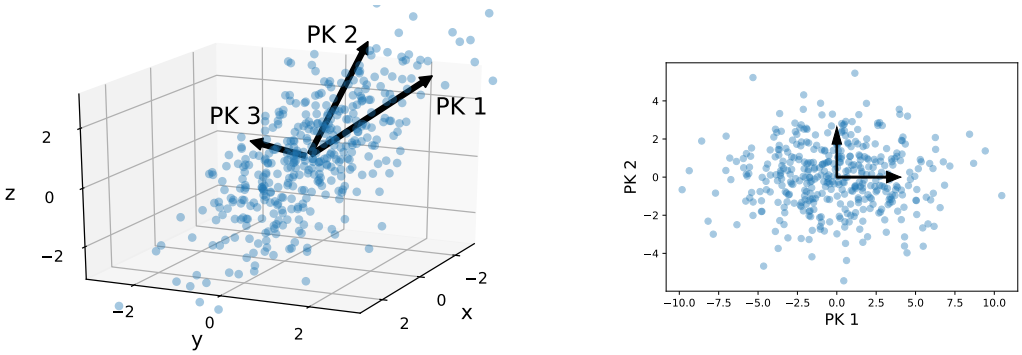
För att kunna arbeta med splines används R-paketet `Splinets` [15]. Paketet använder en speciell ortonormerad bas som benämns *splinet*, ett exempel på en splinet-bas visualiseras i Figur 1.

Projektion av funktioner till splines utförs med funktionen `project()` från paketet. För att funktionen skall kunna utföra projektionen behöver man ange ordningen och noderna till splinerummet man önskar projicera på. Funktionen fungerar också när indatan är en diskretisering av någon kontinuerlig funktionell data. I vårt fall använder vi `project()` för att omvandla vektorer av pixeldata till splines. Den resulterande splinen från `project()` uttrycks i en splinetbas av den specificerade ordningen och över de specificerade noderna.

Ytterligare funktioner som används från Splinets-paketet är `evspline()` och `lincomb()` som evaluerar en spline i de angivna punkterna respektive beräknar en linjärkombination av splines. Den senare funktionen är speciellt användbar då vi skall utföra basbyten i principalkomponentanalys.

2.2 Principalkomponentanalys

Principalkomponentanalys (PCA) är en metod för att reducera datans dimensioner genom att extrahera relevanta variabler som förklarar det mesta av variationen i datan. Principalkomponenterna är en ortogonal bas av vektorer där den första basvektorn pekar i riktningen där datan varierar mest. Den andra vektorn är ortogonal mot den första och kan sägas peka i riktningen där datan varierar näst mest. Idén med PCA är att använda ett fåtal principalkomponenter för att representera datan utan att förlora för mycket information [16].



Figur 2: Visualisering av PCA där tre-dimensionell data reduceras till två dimensioner. Principalkomponenterna har förstorats för att underlätta synligheten.

Mer formellt förklarat: Den första principalkomponenten $\phi_1 = (\phi_{11} \ \dots \ \phi_{p1})^T$ av de stokastiska variablerna $X_1, \dots, X_p$ är den linjärkombination av $X_1, \dots, X_p$ som har maximal varians:

$$\max_{\phi_{11},\dots\phi_{p1}} \text{Var}[Z_1] = \max_{\phi_{11},\dots\phi_{p1}} \text{Var}[\phi_{11}X_1 + \phi_{21}X_2 + \dots + \phi_{p1}X_p]. \tag{2}$$

För att optimeringsproblemet skall vara begränsat kräver vi att vektorn $\phi_1 = (\phi_{11} \ \dots \ \phi_{p1})^T$ är normaliserad. Den andra principalkomponenten är den normaliserade linjärkombination av $X_1, \dots, X_p$ som har maximal varians under bivillkoret att den är okorrelerad med den första principalkomponenten. Bivillkoret att de två komponenterna skall vara okorrelerade är ekvivalent med att de två vektorerna ϕ_1 och ϕ_2 är ortogonala.

Vidare betraktar vi hur principalkomponenterna kan skattas från observerad data. Antag att vi har n observationer av de stokastiska variablerna $X_1, X_2, \dots, X_p$. Låt $n \times p$ matrisen $\mathbf{X}$ innehålla de n observationer av variablerna $X_1, \dots, X_p$. Antag att kolumnerna i matrisen har medelvärde 0 (om detta inte uppfylls subtraheras de kolumnvisa medelvärdena från $\mathbf{X}$). För att skatta den första principalkomponenten betraktar vi linjärkombinationer av observationerna:

$$z_{i1} = \phi_{11}x_{i1} + \dots + \phi_{p1}x_{ip}, \quad i = 1, \dots, n. \tag{3}$$

Att kolumnerna i $\mathbf{X}$ har medelvärde 0 är ekvivalent med att $\sum_{i=1}^{n} x_{ij} = 0$. Därmed gäller att stickprovsvariansen för Z_1 är

$$\frac{1}{n}\sum_{i=1}^{n} z_{i1}^2 = \frac{1}{n}\sum_{i=1}^{n} \left(\phi_{11}x_{i1} + \dots + \phi_{p1}x_{ip}\right)^2. \tag{4}$$

Den första principalkomponenten är den normaliserade vektorn ϕ_1 som maximerar stickprovsvariansen i Ekvation (4) [16]. Enligt [17], gäller att de M första principalkomponenterna är ekvivalenta med de M egenvektorer till kovariansmatrisen av X som tillhör de M största egenvärdena. Därmed kan vi utföra en spektraldekomposition av kovariansmatrisen till X för att få fram en uppsättning egenvektorer och sortera de efter egenvärdenas storlek för att finna principalkomponenterna.

Låt $P =$ `eig(cov(X))` beteckna $p \times p$-matrisen med normerade principalkomponenter där `cov()` beräknar kovariansmatrisen och `eig()` utför en spektraldekomposition samt sorterar egenvektorerna. Vi använder P för att utföra ett basbyte till principalbasen: $\mathbf{X}_p := \mathbf{X}P$. Matrisen $\mathbf{X}_p$ har storleken $n \times p$ precis som $\mathbf{X}$. Vi kan nu välja att betrakta enbart de M första kolumnerna i $\mathbf{X}_p$. Enligt [17] minimerar denna M-dimensionella representation det genomsnittliga kvadratavvikelsen av datan.

2.2.1 Funktionell principalkomponentanalys

Funktionell principalkomponentanalys (FPCA) är en metod för att utforska variationen i observerad funktionell data [6]. I detta avsnitt är vårt mål att från definitionen av FPCA visa att beräkningarna av de funktionella principalkomponenterna kan utföras på samma sätt som i PCA.

I PCA betraktade vi en linjärkombination av stokastiska variabler. Notera att en linjärkombination kan betraktas som en inre produkt av två vektorer. För funktioner motsvaras den inre produkten av integralen: $\int f(x)g(x)dx$ [6]. Den första funktionella principalkomponenten till den stokastiska processen $S(x)$ definieras därför som den funktion $P_1(x)$ som maximerar följande:

$$\max_{P_1(x)} \text{Var}\left[\langle P_1, S\rangle\right] = \max_{P_1(x)} \text{Var}\left[\int_a^b P_1(x)S(x)dx\right], \tag{5}$$

med bivillkoret $||P_1||^2 = \int (P_1(x))^2 dx = 1$. Resterande principalkomponenter definieras på samma sätt med det extra bivillkoret att de är ortogonala mot tidigare principalkomponenter [6]. Antag att vi har n observationer av den stokastiska processen $S(x)$: $s_i(x)$, $i = 1, \ldots, n$. Antag även att $S(x), P_1(x) \in V$ där V är ett linjärt funktionsrum och $\dim(V) = N < \infty$. Låt $\phi_j(x)$, $j = 1, \ldots, N$ beteckna en ortonormerad bas i V så att $s_i(x) = \sum_{k=1}^{N} c_{ik}\phi_k(x)$ och $P_1(x) = \sum_{j=1}^{N} \xi_{1j}\phi_j(x)$. Stickprovsvariansen av $\langle P_1, S\rangle$ är då ekvivalent med:

$$\frac{1}{n}\sum_{i=1}^{n}\langle P_1, s_i\rangle^2 = \frac{1}{n}\sum_{i=1}^{n}\Big(\int_a^b P_1(x)s_i(x)dx\Big)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(\int_a^b \Big(\sum_{j=1}^{N}\xi_{1j}\phi_j(x)\Big)\Big(\sum_{k=1}^{N}c_{ik}\phi_k(x)\Big)dx\right)^2 = \tag{6}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\Big(\sum_{j=1}^{N}\xi_{1j}c_{ij}\int_a^b \phi^2(x)dx\Big)^2 = \frac{1}{n}\sum_{i=1}^{n}\Big(\sum_{j=1}^{N}\xi_{1j}c_{ij}\Big)^2. \tag{7}$$

Likheterna följer av att $\phi_j(x)$, $j = 1, \ldots, N$ är en ortonormerad bas. Notera att det slutgiltiga uttrycket för stickprovsvariansen i princip är lika med uttrycket för den diskreta stickprovsvariansen i ekvation (4). Genom att betrakta koefficienterna c_{ij} istället för de observerade processerna $s_i(x)$ noterar vi att beräkningarna i FPCA kan utföras på samma sätt som i PCA. Detta håller enbart om $S(x), P(x) \in V$ och $\dim(V) = N < \infty$ uppfylls. Genom att låta V vara ett funktionsrum av splines med en specifik ordning och val av knots kan vi uppfylla dessa krav.

Observationerna s_i kan approximeras med de M första principalkomponenterna enligt följande: $s_i^M(x) = \sum_{j=1}^{M} b_{ij}P_j(x)$ där $b_{ij} = \int s_i(x)P_j(x)dx$. Enligt [6], gäller att de M första principalkomponenterna är den M-dimensionella bas som minimerar den totala kvadratavvikelsen av observationerna:

$$\sum_{i=1}^{n}||s_i - s_i^M||^2 = \sum_{i=1}^{n}\int_a^b (s_i(x) - s_i^M(x))^2 dx. \tag{8}$$

I senare delar av arbetet förklarar vi hur ovanstående kan användas för klassificering.

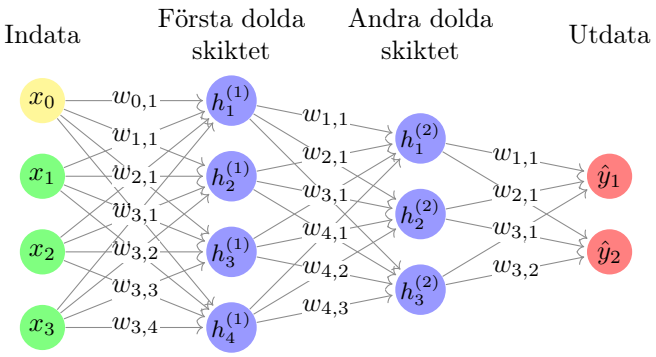
2.3 Artificiella neuronnät

Artificiella neuronnät [18] är algoritmer som härmar beteendet hos den mänskliga hjärnan genom datorstödda beräkningar. Det är baserat på noder, även kallat neuroner, som skickar signaler till varandra, där signalerna är reella tal. En nod tar emot signaler från intilliggande noder och får med summan av dessa signaler och en *aktiveringsfunktion*, ett värde som den skickar vidare till nästa nod. Det går att betrakta noderna i ett neuronnät som olika lager. Då tas signalen först emot av det första laget, inmatningslagret, och vandrar sedan igenom neuronnätet till det sista lagret, utgångslagret. Matematiskt representeras noderna av icke-linjära funktioner $f^{(i)}$ som binds samman med en operator för att bilda det artificiella neuronnätet f. Den vanligaste strukturen hos artificiella neuronnät är faltning, som då bildar en kedja med olika lager. För att estimera parametrarna för de olika lagren i ett neuronnät används en träningsfas som görs på en del av datan. Resterande del av datan används för att testa det upptränade neuronnätet. I träningsfasen

så jämför vi det vi får ut av neuronnätet, $f(x)$, för olika inmatningsvärden x med $y \approx f^*(x)$ där $f^*(x)$ är de sanna värdena för inmatningsvärdena.

En viktig uppsättning av klassificeringssmetoder är baserade på neuronnät. Artificiella neuronnät eller ANN används i många olika sammanhang såsom mönsterigenkänning, prognoser och klassificering. Nätens arkitektur anpassas till de uppgifter den är avsedd att ta itu med som exempelvis klassificering, optimering eller regression. Ett populärt nätverk är det så kallade *fully connected* nätverket, se figur 3. Här ingår även olika skikt av neuroner, linjära klassificerare, med inlärningsalgoritmen *Backpropagation* [18] vilka är användbara för uppgiftstyper av både klassificering och regression.

Det är viktigt att poängtera att nätverkets mål är att minimera förlustfunktionen $(f(x) - y)^2$ genom att optimera vikterna $\omega_{i,j}$. När vikterna, $\omega_{i,j}$, har framtagits och uppsättningen av dem har fått en tillfredsställande prestation så är vi redo för klassificering.



Figur 3: Ett fullt anslutet nätverk med två dolda lager av neuroner.

2.3.1 Fullt anslutet neuronnät

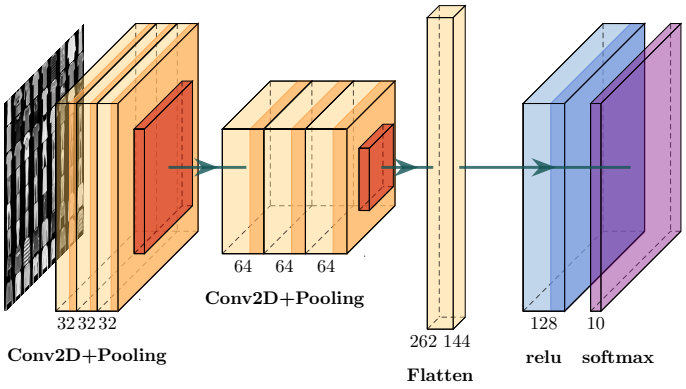
Fullt anslutet neuronnät (FNN) är en typ av artificiella neuronnät med en arkitektur sådan att alla neuroner i ett lager är anslutna till alla neuroner i nästa lager. I figur 3 har vi ett exempel på ett FNN. Den största fördelen med fullt anslutna neuronnät är att de är *agnostiska datastrukturer* vilket innebär att det behövs inga speciella antaganden om indatan [19].

För att kunna beskriva hur *arkitekturen* av ett fullt anslutet neuronnät varierar i senare delar av arbetet introducerar vi notation för antalet noder i varje lager. I det generella fallet beskriver vi ett fullt anslutet neuronnät som: $[I, n_1, n_2, \ldots, n_L, O]$ där I anger antalet dimensionen i indatan, $n_i \in \mathbb{N}$ anger antal noder i lager i, L anger antalet dolda lager i nätverket och O anger antalet dimensioner i utdatan. Nätverket i figur 3 uttrycks enligt notationen då som [4,4,3,2].

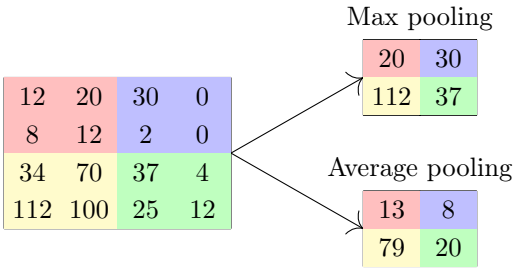
2.3.2 Faltande neuronnät

Ett faltande neuronnät (CNN[1]) är en speciell implementering av ett neuronnät som behandlar uteslutande matrisdata såsom bilder. Dess arkitektur antar uttryckligen att indatan är bilder, vilket möjliggör kodning av vissa egenskaper i modellarkitekturen. Se Figur 4 för en visualisering av ett CNN. Ett enkelt CNN är en sekvens av lager, och varje lager omvandlar ett antal aktiveringar till en annan genom en differentierbar funktion. För att skapa arkitekturen till ett faltande neuronnät används tre huvudtyper av lager, poolning-lager, faltningslager och fullt anslutna lager [19]. För poolning-lagret så används främst följande två typer av poolning: max-poolning och medelvärdespoolning. I figur 5 ser vi ett exempel på hur poolning har transformerat en $4x4$ bild till en $2x2$ bild. Max-poolning hjälper till att ta bort brus i datan samtidigt som det reducerar dimensioner. Medelvärdes-poolning fungerar endast som dimensionsreducerare och därför föredras ofta max-poolning över medelvärdes-poolning [5].

[1]Från det engelska namnet Convolutional Neural Network.



Figur 4: Visualisering av ett CNN



Figur 5: Två typer av pooling

Ett faltningslager använder matematisk faltning för att fånga distinkta objekt som exempelvis en kant i en bild. Faltningslagret gör detta genom att transformera indatan på liknande sätt som för poolning. Vi kan använda flera faltningslager för att fånga mindre distinkta objekt och på så sätt göra nätverket mer intelligent [5].

2.3.3 Implementering av ANN

För att implementera de neurala nätverk som beskrivs ovan använder vi oss av det välkända maskininlärningspaketet *Keras* [10]. Keras ges oss möjlighet att på ett smidigt sätt konfigurera, kompilera och träna nätverken samt extrahera information om noggrannheten.

För att bygga ett FNN i Keras behöver vi specificera antalet lager i modellen, antalet noder i varje lager samt storleken på indatan och utdatan. I figur 6 anges koden som krävs för att bygga, kompilera samt anpassa ett neuronnät med arkitekturen [784,128,10]. Ytterligare parametrar som anges till neuronnätet är aktiveringsfunktioner, regulariseringsmetod, optimeringsalgoritm, förlustfunktion (objektivfunktion) samt antalet epoker. Innebörden av dessa ytterligare parametrar och deras specificerade värden redogörs i ordlistan i appendix A. Med undantag för regulariseringsmetod så används samtliga parametrar så som de är angivna i figur 6 genom hela arbetet.

2.4 Val av hyperparametrar med hjälp av korsvalidering

För att välja parametrar så som neuronnätens arkitektur eller antal relevanta principalkomponenter används metoden korsvalidering. Med korsvalidering uppskattas en modells prediktionsfel på icke-observerad data genom att dela upp datan i K jämna delar och upprepat låta modellen enbart ha tillgång till en delmängd av datan [20]. Låt $\mathcal{D}_i$ beteckna den i:te delen av datan för $i = 1, \ldots K$. I K-delad korsvalidering upprepas följande för $i = 1, \ldots, K$:

1. Åsidosätt $\mathcal{D}_i$.

```
model=keras.Sequential([
        keras.layers.Flatten(input_shape=(28,28)),
        keras.layers.Dense(128, activation='relu', kernel_regularizer = 'l2'),
        keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=10)
```

Figur 6: Konfigurering och kompilering samt träning av ett neuralt nätverk med Keras.

2. Träna din modell på resterande data: $\bigcup_{j=1,j\neq i}^{K} \mathcal{D}_j$.
3. Beräkna modellens prediktionsfel på den åsidosatta datan, låt $\mathcal{E}_i$ beteckna prediktionsfelet på $\mathcal{D}_i$.

Slutligen summeras prediktionsfelen till ett genomsnittligt prediktionsfel $\mathcal{E} = \sum_{i=1}^{K} \mathcal{E}_i/\mathrm{K}$. Med hjälp av K-delad korsvalidering kan man uppskatta olika modellers prediktionsfel och sålla fram de bästa modellerna enbart med träningsdatan.

Men hur bör man välja K? Höga värden av K kan kräva mycket beräkningstid då modellen behöver återanpassas många gånger. Låga värden av K kan överskatta prediktionsfelet om storleken på den åsidosatta datan förhindrar modellen att anpassa sig till datan. En bra kompromiss är att låta K vara antingen 5 eller 10 [20].

Modellerna vi använder har flera parametrar som behöver bestämmas innan modellerna anpassas till datan. Dessa benämns ofta som hyperparametrar och inkluderar i FPCA: antalet noder i spline-representationen, ordningen på spline-representationen samt antalet principalkomponenter att använda. Hyperparametrar för neuronnät inkluderar arkitekturen av neuronnätet och valet av optimeringsalgoritm (som ofta har egna hyperparametrar).

3 Dataset och databehandling

I detta arbete använder vi det publikt tillgängliga datasetet Fashion-MNIST [21]. Fashion-MNIST (datan) innehåller totalt 70,000 unika bilder av klädesplagg från online-återförsäljaren Zalando. Bilderna har reducerats i storlek till 28×28 pixlar och omvandlats till gråskala [21]. Datan har delats upp i två delar, träningsdata med 60,000 bilder och testdata med 10,000 bilder. Träningsdatan används för att träna en modell att klassificera data medan testdatan används för att säkerställa att modellen kan klassificera data den inte tidigare har observerat [16]. Denna metodik gör att vi kan avgöra om modellen generaliserar träningsdatan till testdatan eller om den enbart kommer ihåg träningsdatan.

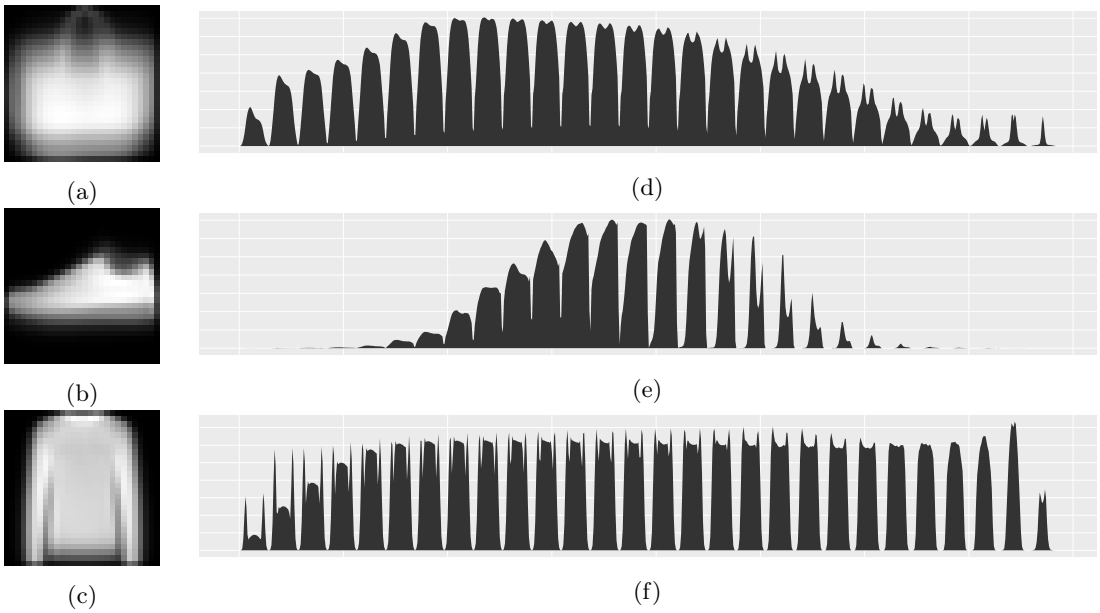
Tabell 1 visar exempel på de 10 klasserna i datan. Datan innehåller 6,000 bilder av varje klass. Bilderna är lagrade som 28×28 matriser med värden i intervallet $[0, 1]$. Matriserna har omvandlats till $28^2 \times 1$ vektorer för att göra datan endimensionell och är visualiserad i graferna i Tabell 1. Detta görs för att funktionell dataanalys är lättare att tillämpa på endimensionell data. Bilderna och vektorerna återges även efter att medelvärdet för klassen subtraherats vilket är nödvändigt för att tillämpa principalkomponentanalys, se avsnitt 2.2. I Figur 7, ser vi en- samt tvådimensionella representationer av det genomsnittliga klädesplagget för tre klasser. Vid jämförelse av bilderna i Tabell 1 och Figur 7 observerar vi att bilderna har jämnats ut och förlorar vissa detaljer men har kvar den generella formen av sina klasser.

3.1 Framtagning av ordinära respektive klassvisa principalkomponenter

I denna sektion förklaras hur ordinära principal- och klassvisa principalkomponenter erhålles för detta dataset. Återigen, bilderna behandlas som vektorer med 28^2 element istället för 28×28-matriser. Låt $\mathbf{v}_i \in \mathbb{R}^{28^2}$ beteckna vektorn för bild i och $\mathbf{D} \in \mathbb{R}^{60\,000\times28^2}$ beteckna matrisen med

Tabell 1: Klasserna i Fashion-MNIST [21] med deras engelska beskrivning samt exempel. Exemplen återges i dess ursprungliga form samt subtraherad med medelvärdet för klassen.

Klass	Beskrivning	Exempel	Klass	Beskrivning	Exempel
0	T-shirt/ Top		5	Sandals	
1	Trouser		6	Shirt	
2	Pullover		7	Sneaker	
3	Dress		8	Bag	
4	Coat		9	Ankle boots	



Figur 7: Genomsnitt av alla klädesplagg av typ Bag, Sneaker samt Pullover. Visualiserat tvådimensionellt i a), b) och c) samt endimensionellt i d), e), f).

träningsdatan där $\mathbf{v}_i$ finns på rad i. För att tillämpa funktionell principalkomponentanalys omvandlas datan till splines av ordning $O = 4$ med $K = 224$ knots jämt fördelade mellan 1 och 28^2. Bilden i representeras då av en spline $f_i(x)$ istället för en vektor med 28^2 element. Funktionen `project` från Splinets-paketet uttrycker splinen f_i i en ortonormerad bas:

$$f_i(x) = \sum_{j=1}^{N} c_{ij} S_j(x),$$

där $\{S_j\}_{j=1}^{N}$ är basfunktionerna och antalet funktioner i basen ges av: $N = K - O - 1$. Låt $\mathbf{C}$ beteckna matrisen av koefficienter till bildernas Spline-representation där element $C_{ij} = c_{ij}$. Ordinär och klassvis PCA utförs sedan på datamatrisen $\mathbf{D}$ och koefficientmatrisen $\mathbf{C}$. Ordinär PCA utförs på samma sätt som beskrivs i avsnitt 2.2 men vi förklarar nedan mer detaljerat hur både ordinär och klassvis PCA utförs.

3.1.1 Ordinär principalkomponentanalys av datan

Vi betraktar här hur ordinär principalkomponentanalys utförs på koefficientmatrisen $\mathbf{C}$ (FPCA-H), motsvarande fungerar på samma sätt för datamatrisen $\mathbf{D}$ (PCA-H). Låt elementen i vektorn $\bar{\mathbf{c}}$ vara de kolumnvisa medelvärdena av $\mathbf{C}$. Vektorn $\bar{\mathbf{c}}$ representerar koefficienterna för den genomsnittliga Splinen av hela datasetet. Vi subtraherar $\bar{c}$ från varje rad i $\mathbf{C}$ och definierar den centrerade koefficientmatrisen:

$$\bar{\mathbf{C}} := \mathbf{C} - \big[1, \ldots, 1\big]^T \cdot \bar{\mathbf{c}}^T. \tag{9}$$

Notera att kolumnerna i $\bar{\mathbf{C}}$ har medelvärde 0 vilket tillåter oss att utföra principalkomponentanalys på $\bar{\mathbf{C}}$. Låt $\mathbf{P}_C$ beteckna matrisen med normerade principalkomponenter till $\bar{\mathbf{C}}$ där j:te kolumnen i $\mathbf{P}_C$ innehåller den j:te principalkomponenten. Vi använder $\mathbf{P}_C$ till att utföra ett basbyte: $\mathbf{C}_P := \bar{\mathbf{C}}\mathbf{P}_C$. Med detta basbyte kan vi approximera f_i som en linjär kombination av M principalkomponenter:

$$f_i(x) \approx f_i^M(x) = \sum_{j=1}^{N} (\mathbf{C}_P)_{ij} P_j(x) + \sum_{j=1}^{N} \bar{\mathbf{c}}_j S_j(x), \tag{10}$$

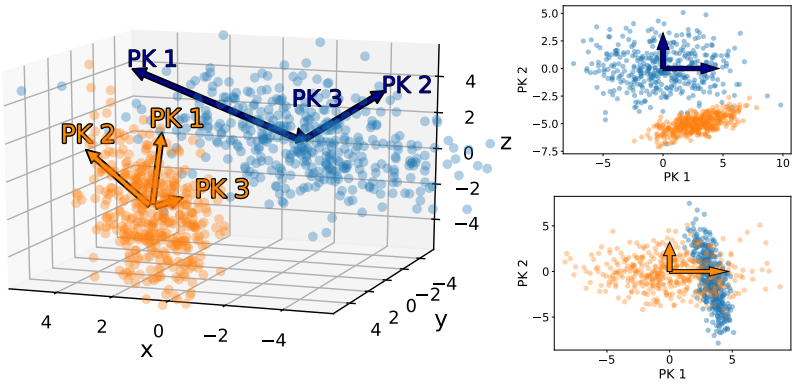
där $P_j(x)$ är funktionen för den j:te principalkomponenten och ges av $P_j(x) = \sum_{k=1}^{N} (\mathbf{P}_C)_{kj} S_k(x)$. Notera att den andra summan enbart är medelvärdet av datasetet och kommer från att vi centrerar datan. Dessutom gäller att $f_i(x) = f_i^M$ då $M = N$. För att koefficienterna $(\mathbf{C}_P)_{ij}$ skall vara av liknande storleksordning för $j = 1, \ldots, N$ dividerar vi varje kolumn i $\mathbf{C}_P$ med standardavvikelsen för kolumnen. Standardavvikelserna för kolumnerna är lika med kvadratroten av egenvärdena till kovariansmatrisen av $\bar{\mathbf{C}}$ och benämns därför som egenvärden. Låt λ_C^j beteckna egenvärdet tillhörande den j:te principalkomponenten och $\hat{\mathbf{C}}_P$ beteckna matrisen med normaliserade koefficienter. Att koefficienterna är normaliserade innebär att kolumnerna i $\hat{\mathbf{C}}_P$ har medelvärde 0 och varians 1. Funktionen f_i^M kan nu uttryckas som:

$$f_i^M(x) = \sum_{j=1}^{M} (\hat{\mathbf{C}}_P)_{ij} \sqrt{\lambda_C^j} P_j(x) + \sum_{j=1}^{N} \bar{\mathbf{c}}_j S_j(x). \tag{11}$$

För datamatrisen $\mathbf{D}$ kan vi på samma sätt approximera vektorn för bild i med M principalkomponenter:

$$\mathbf{v}_i \approx \mathbf{v}_i^M = \sum_{j=1}^{M} (\hat{\mathbf{D}}_P)_{ij} \sqrt{\lambda_D^j} P_j + \bar{\mathbf{d}}, \tag{12}$$

där P_j är den j:te principalkomponenten (numera en vektor) och $\bar{\mathbf{d}}$ är vektorn med kolumnvisa medelvärden av $\mathbf{D}$.



Figur 8: Klassvis principalkomponentanalys där simulerad data projiceras på principalkomponenter för båda klasserna. Principalkomponenterna har förstorats för att underlätta synligheten.

3.1.2 Klassvis principalkomponentanalys av datan

Figur 8 visar resultatet av klassvis PCA där datapunkter projiceras på principalkomponenter för de två klasserna. I detta avsnitt beskriver vi hur klassvis utförs på koefficientmatrisen $\mathbf{C}$ (FPCA-KV), motsvarande har även utförts på datamatrisen $\mathbf{D}$ (PCA-KV). Målet är att emulera det som sker i figur 8 i högre dimensioner och med fler klasser. Beräkningarna liknar ordinär PCA förutom att vi behöver särskilja vilka rader i matrisen $\mathbf{C}$ som tillhör vilken klass och upprepa metoden för ordinär PCA.

Låt $\mathbf{C}^k$ beteckna matrisen med koefficienter för bilder av klass $k \in \{0, 1, \ldots, 9\}$ och låt vektorn $\bar{\mathbf{c}}^k$ innehålla de kolumnvisa medelvärdena av $\mathbf{C}^k$. Vektorn $\bar{\mathbf{c}}^k$ representerar koefficienterna för den genomsnittliga Splinen av klass k. Vi subtraherar $\bar{\mathbf{c}}^k$ från varje rad av $\mathbf{C}^k$, utför principalkomponentanalys på den resulterande matrisen och får en $N \times N$-matris med normerade principalkomponenter till klass k: $\mathbf{P}_C^k$. Vi subtraherar sedan $\bar{\mathbf{c}}^k$ från varje rad i koefficientmatrisen $\mathbf{C}$ och utför ett basbyte till principalkomponenterna för klass k: $\mathbf{C}_P^k := (\mathbf{C} - \begin{bmatrix}1, \ldots, 1\end{bmatrix}^T \cdot (\bar{\mathbf{c}}^k)^T) P_C^k$. Raderna i matrisen $\mathbf{C}_P^k$ motsvarar specifika koordinater för datapunkterna uttryckta i principalkomponentbasen för klass k, ett exempel av detta visualiseras till höger i figur 8. Vi dividerar kolumnerna i $\mathbf{C}_P^k$ med standardavvikelsen för kolumnerna och får en matris med normaliserade koefficienter: $\hat{\mathbf{C}}_P^k$. Med detta kan vi approximera funktionen f_i med M principalkomponenter för klass k:

$$f_i(x) \approx f_i^{M,k}(x) = \sum_{j=1}^{M} (\hat{\mathbf{C}}_P^k)_{ij} \sqrt{\lambda_C^{k,j}} P_j^k(x) + \sum_{j=1}^{N} \bar{\mathbf{c}}_j^k S_j(x) \tag{13}$$

där $P_j^k(x)$ är den j:te principalkomponenten till klass k och $\lambda_C^{k,j}$ är egenvärdet tillhörande $P_j^k(x)$. För datamatrisen $\mathbf{D}$ kan vi enligt samma metod approximera vektorn $\mathbf{v}_i$ av en bild i med hjälp av M principalkomponenter för klass k:

$$\mathbf{v}_i \approx \mathbf{v}_i^{M,k} = \sum_{j=1}^{M} (\hat{\mathbf{D}}_P^k)_{ij} \sqrt{\lambda_D^{k,j}} P_j^k + \bar{\mathbf{d}}^k, \tag{14}$$

där vektorn $\bar{\mathbf{d}}^k$ är genomsnittet av samtliga vektorer $\mathbf{v}_l$ som tillhör klass k. Genomsnittsvektorn $\bar{\mathbf{d}}^k$ motsvarar genomsnittsbilden för klass k och exempel visas i figur 7.

3.1.3 Användning av matriserna med normaliserade koefficienter

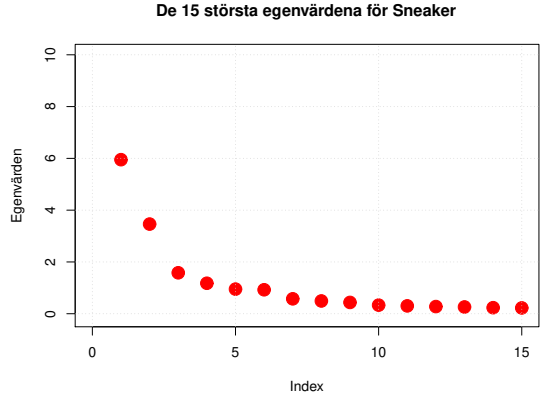
I avsnitt 3.1.1 och 3.1.2 har vi beskrivit ett antal matriser med koefficienter som i avsnitt 4 används för att klassificera bilder. Här ges en kort sammanfattning av vilka matriser vi har tagit fram. För den ordinära PCA:n har vi $\hat{\mathbf{C}}_P$ respektive $\hat{\mathbf{D}}_P$ som innehåller normaliserade koefficienter för den funktionella principalkomponentbasen respektive principalkomponentbasen med vektorer.

Från den klassvisa PCA:n har vi tagit fram motsvarande matriser för varje klass k: $\{\hat{\mathbf{C}}_P^k\}_{k=0}^9$ samt $\{\hat{\mathbf{D}}_P^k\}_{k=0}^9$. I den klassvisa PCA:n med M principalkomponenter approximeras bild i upprepade gånger av funktionerna: $f_i^{M,k}(x)$, $k \in \{0,1,\ldots,9\}$. Eftersom principalkomponenterna för klass k enbart beräknas med data för klass k förväntas $f_i^{M,k}(x)$ enbart approximera f_i väl om bild i tillhör klass k.

3.2 Analys av funktionella principalkomponenter för klassen Sneaker

I denna del redovisas funktionella principalkomponenter till klassen Sneaker samt approximationerna $f_i^{M,k}$ från ekvation (13), med $k = 7$ för klassen Snaker. Kom ihåg att en principalkomponent är en egenfunktion med ett egenvärde från kovariansmatrisen som motsvarar principalkomponentens riktning och storlek, se sektion 2.2.

I figur 9 visas de 15 största egenvärdena $\lambda_C^{7,1},\ldots,\lambda_C^{7,15}$ som tillhör de 15 första principalkomponenterna för klassen Sneaker. De egenvärden med index större än 10 är nära 0 och kan ge en fingervisning av hur stort M bör vara för att $f_i^{M,k}$ skall approximera f_i väl. I bilaga C visas egenvärden för samtliga klasser där vi ser att egenvärdena avtar i liknande takt.



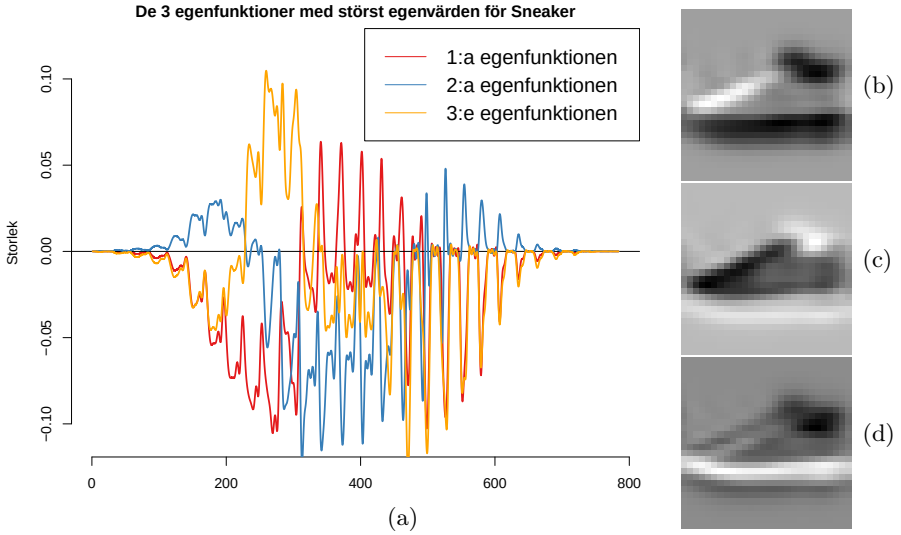
Figur 9: De 15 största egenvärdena till klassen Sneaker: $\lambda_C^{7,1},...,\lambda_C^{7,15}$. Storleken avtar med ökande index.

Fortsättningsvis visualiseras de tre första egenfunktionerna för klassen Sneaker ($P_1^7(x)$, $P_2^7(x)$, $P_3^7(x)$) i figur 10a. De tre funktionerna har omvandlats till bilder som visas i figurerna 10b, 10c och 10d. Omvandlingen till bilder sker genom att evaluera funktionerna i punkterna $1,2,\ldots,28^2$, omvandla de 3 resulterande vektorerna med 28^2 element vardera till tre 28×28-matriser och sedan använda funktionen `image` i R. Bilderna av funktionerna $P_1^7(x)$, $P_2^7(x)$ och $P_3^7(x)$ liknar var för sig en Sneaker. Figurer med motsvarande bilder av de första funktionella principalkomponenterna för samtliga klasser, $P_1^k(x)$, visas i bilaga D.

3.3 Projektion av en Sneaker och en T-shirt på egenfunktionerna för klassen Sneaker

I detta avsnitt illustreras en idé på hur de funktionella principalkomponenterna kan användas för att klassificera bilder och vad de funktionella principalkomponenterna potentiellt kan säga om de olika klasserna.

Låt f_S och f_T beteckna funktionen för en Sneaker respektive en T-shirt hämtade från datasetet. Funktionerna f_S och f_T visas i figur 11a respektive 12a. Vi påminner om att $f_S^{M,7}$ och $f_T^{M,7}$ är projektionerna av f_S och f_T på de M första principalkomponenterna för klassen Sneaker, adderat



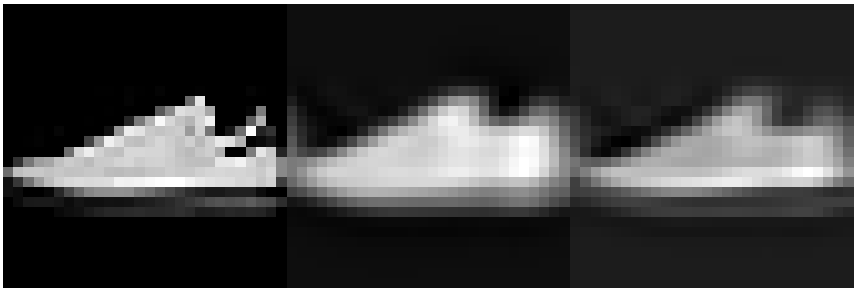
Figur 10: De 3 egenfunktioner $P_1^7(x)$,$P_2^7(x)$,$P_3^7(x)$ med störst egenvärden för klassen Sneaker som splines i (a) och omvandlade till bilder i (b), (c) och (d).

med genomsnittliga splinen för Sneaker:

$$f_S^{M,7}(x) = \sum_{j=1}^{M} (\hat{\mathbf{C}}_P^7)_{Sj} \sqrt{\lambda_C^{7,j}}\, P_j^7(x) + \sum_{j=1}^{N} \bar{\mathbf{c}}_j^7 S_j(x), \quad f_T^{M,7}(x) = \sum_{j=1}^{M} (\hat{\mathbf{C}}_P^7)_{Tj} \sqrt{\lambda_C^{7,j}}\, P_j^7(x) + \sum_{j=1}^{N} \bar{\mathbf{c}}_j^7 S_j(x). \tag{15}$$

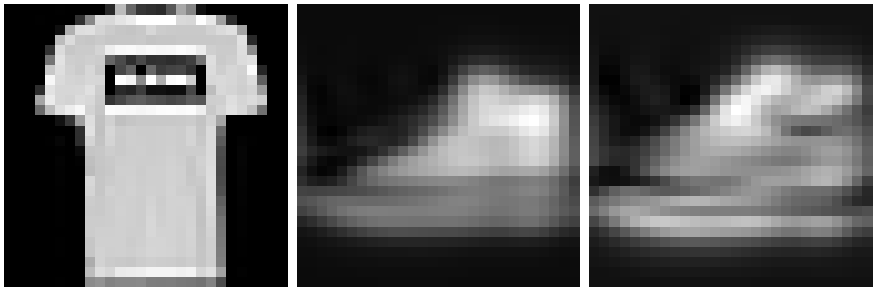
Figur 11 visar bilderna av de tre funktionerna f_S, $f_S^{M,7}$ där $M = 3, 10$ och figur 12 visar bilderna av de tre funktionerna f_T, $f_T^{M,7}$ där $M = 3, 10$. Från figur 12 ser vi att projektionerna av T-shirt:en blir otydlig och inte alls lik originalbilden f_T, jämfört med projektionerna av Sneaker:n i figur 11 som liknar orginalbilden f_S för båda projektioner. Dessutom är projektionen av T-shirt:en otydligare för $M = 10$ jämfört med $M = 3$.

Genom att projicera en bild av okänd klass på principalkomponenter för de olika klasserna och jämföra originalbilden mot projektionen kan detta användas för att klassificera. Detta är idén för klassificering med klassvisa principalkomponenter och den exakta metoden tas upp under sektion 4.2.1 och 4.2.2.



(a) (b) (c)

Figur 11: Ett exempel på en Sneaker f_S visas i (a). I (b) och (c) visas projektionen av denna Sneaker, $f_S^{M,7}$, på $M = 3$ respektive $M = 10$ principalkomponenter för klassen Sneaker. Notera att projektionerna liknar original-Sneakern i (a).



(a) (b) (c)

Figur 12: Ett exempel på en T-shirt f_T visas i (a). I (b) och (c) visas projektionen av denna T-shirt, $f_T^{M,7}$, på $M = 3$ respektive $M = 10$ st egenfunktioner för klassen Sneaker. Notera att projektionerna $f_T^{M,7}$ inte liknar original-T-shirten i (a) och dessutom att projektionen på $M = 10$ egenfunktioner verkar suddigare än projektionen då $M = 3$.

4 Metod

I denna sektion förklaras metoderna som använts för att klassificera datan samt hur vi använt korsvalidering för att välja värden på hyperparametrar i metoderna. Inledningvis beskrivs en enkel klassificeringsmetod baserat på avstånd till klassernas medelvärden i avsnitt 4.1. Vidare beskrivs flera metoder att klassificera baserat på principalkomponentanalys i avsnitt 4.2 och slutligen beskrivs arkitekturen på neuronnäten som har använts.

Samtliga metoder som undersökts har någon form av hyperparameter vars värde måste bestämmas på förhand. Vad värdet av dessa hyperparametrar bör vara är svårt att veta utan att testa metoderna på flera olika värden. För metoderna i avsnitt 4.2.3 samt 4.3 används korsvalidering med $K = 10$ för att välja ut antalet principalkomponenter i 4.2.3 respektive arkitektur för ANN i 4.3. För metoderna i avsnitt 4.1, 4.2.2 samt 4.2.1 används inte korsvalidering då nyttan av korsvalidering bedömts inte väga upp de extra beräkningstiderna som tillkommer. Slutligen, bör det även noteras att beräkningarna av principalkomponenter som beskrevs i avsnitt 3.1 endast genomfördes en gång men att korsvalidering genomfördes på den resulterande datan från PCA.

För att jämföra hur väl modellerna klassificerar bilderna använder vi måttet noggrannhet som vi definierar som andelen korrekta klassifikationer utav samtliga klassifikationer. Måttet noggrannhet lämpar sig väl då datasetet innehåller lika många bilder av varje klass.

4.1 Klassificering med medelvärde

I detta avsnitt beskriver vi en enkel metod för att klassificera bilderna i datasetet. Metodens resultat används för att etablera en referens som vi kan jämföra de mer komplexa modellerna emot. För att klassificera bild i beräknar vi det Euklidiska avståndet från bild i till medelvärdet av samtliga klasser:

$$\text{dist}(\mathbf{v}_i,\bar{\mathbf{d}}^k) = ||\mathbf{v}_i - \bar{\mathbf{d}}^k||_2 \quad k \in \{0,1,\ldots,9\}. \tag{16}$$

Bild i klassificeras sedan som den klass k som minimerar $\text{dist}(\mathbf{v}_i,\bar{\mathbf{d}}^k)$. Givet att bilder av samma klass ligger nära varandra spatialt bör metoden kunna klassificera bilderna väl.

4.2 Klassificering med principalkomponenter

Efter att ordinära respektive funktionella principalkomponenter erhållits i sektion 3.1.1 och 3.1.2 används dessa för klassificering av datan på två olika sätt. Den första metoden går ut på att minimera avståndet mellan spline-/vektorrepresentationen av en bild och dess projektion på de principalkomponenterna. Den andra metoden går ut på att träna ett neuronnät med de ordinära samt klassvisa principalkomponenterna.

4.2.1 Klassificering genom bäst funktionella approximation

Låt $f(x)$ vara en okänd bild som ska klassificeras. Antag att f tillhör någon av klasserna $k = 0, 1, ..., 9$. Vi påminner om ekvation (10), som säger att vi kan approximera spline-representationen av en bild f_i med M principalkomponenter för klass k:

$$f_i(x) \approx f_i^{M,k}(x) = \sum_{j=1}^{M} (\mathbf{C}_P^k)_{ij} P_j^k(x) + \sum_{j=1}^{N} \bar{\mathbf{c}}_j^k S_j(x) \tag{17}$$

Då $M = N$, där N är antalet basfunktioner för spline-basen, gäller $f_i = f_i^{M,k}$ för någon klass k.

Betrakta residualen $\epsilon_k(x) = f_i - f_i^{M,k} = \sum_{j=M+1}^{N} (\mathbf{C}_P^k)_{ij} P_j^k(x)$ som ger differensen mellan spline-representationen av originalbilden och projektionen på M principalkomponenter för en klass k.

För ett mått på residualens storlek används L_2-normen i kvadrat: $||\epsilon_k(x)||_2^2 := \int_{\mathbb{R}} \epsilon_k(x)^2 dx$. Genom att använda att $\{P_j^k\}_{j=1}^M$ är en ON-bas kan uttrycket förenklas till:

$$||\epsilon_k(x)||_2^2 = \sum_{j=M+1}^{N} \left((\mathbf{C}_P^k)_{ij} \right)^2 ||P_k^j||_2^2 = \sum_{j=M+1}^{N} \left((\mathbf{C}_P^k)_{ij} \right)^2 \tag{18}$$

Bilden klassificeras som den klass k som minimerar $||\epsilon_k||_2^2$.

Denna metod testades för $M = 5, 10, 20, 30, 60, 90, 120, 150, 180$. För denna metod går det inte att tillämpa korsvalidering eftersom modellen inte tränas.

4.2.2 Klassificering med bästa vektorapproximation

Denna metod är snarlik metoden beskriven i 4.2.1, fast istället för ekvation (10) används motsvarande ekvation med datamatrisen $(\mathbf{D}_P^k)_{ij}$ som säger att vi kan approximera vektorn $\mathbf{v}_i$ för en bild i med M principalkomponenter för klass k enligt:

$$\mathbf{v}_i \approx \mathbf{v}_i^{M,k} = \sum_{j=1}^{M} (\mathbf{D}_P^k)_{ij} P_j^k + \bar{\mathbf{d}}^k, \tag{19}$$

På samma sätt klassificeras bild i som den klass k som minimerar $||\boldsymbol{\epsilon}_k||_2^2$, där residualen ges av $\boldsymbol{\epsilon}_k = \sum_{j=M+1}^{N} (\mathbf{D}_P^k)_{ij} P_j^k$ och $N = 28^2$ är dimensionen hos $\mathbf{v}_i$. Då principalkomponenterna $\{P_j^k\}_{j=1}^N$ utgör en ortonormerad bas kan $||\boldsymbol{\epsilon}_k||_2^2$ förenklas:

$$||\boldsymbol{\epsilon}_k||_2^2 = \sum_{j=M+1}^{N} \left((\mathbf{D}_P^k)_{ij} \right)^2 \tag{20}$$

Denna metod testades för $M = 5, 10, 20, 30, 60, 90, 120, 150, 180, 250, 350, 500$.

4.2.3 Klassificering med ordinära respektive klassvisa principalkomponenter och neuronnät

I detta avsnitt förklaras hur vi använt neuronnät för att klassificera bilder med hjälp av matriserna som beräknades i avsnitt 3.1.1 och 3.1.2. Ett fullt ansluten neuronnät har använts med två dolda lager där vardera lager har $20 \cdot M$ noder. Lagrena använder L_2-regularisering för vikterna och ReLu som aktiveringsfunktion. Antalet epoker var 10 varje gång ett neuronnät tränades. Korsvalidering har tillämpats med $K = 10$ för att bestämma värdet av M, antalet principalkomponenter.

I varje omgång av korsvalidering tränades separata nätverk med följande värden av M:

$$M \in \{5, 10, 20, 30, 40, 60, 80\}.$$

För varje val av M, tränades fyra kategorier av neuronnät som alla fick olika indata. De två första kategorierna av neuronnät benämner vi PCA-H och FPCA-H för att de använder de första M kolumnerna i $\hat{\mathbf{D}}_P$ respektive $\hat{\mathbf{C}}_P$ som indata. Suffixet -H syftar på att indatan är baserad på principalkomponentanalys som utförs på hela datasetet. De två återstående kategorierna av neuronnät använde de första M kolumnerna från samtliga 10 klassvisa koefficientmatriser: $\left\{\hat{\mathbf{D}}_P^k\right\}_{k=0}^9$ respektive $\left\{\hat{\mathbf{C}}_P^k\right\}_{k=0}^9$. Dessa har vi benämnt som PCA-KV respektive FPCA-KV för att principalkomponentanalys har utförts klassvis på indatan. Den genomsnittliga klassificeringsnoggrannheten vid korsvalidering för de fyra kategorierna av neuronnät redovisas i tabell 3.

4.3 Klassificering med neuronnät

I detta avsnitt beskrivs hur vi använt neuronnät av typerna FNN och CNN för att klassificera bilderna i datan baserat på dess vektorrepresentation respektive matrisrepresentation. Sex olika arkitekturer av båda typerna av neuronnät utvärderades genom att använda korsvalidering med $K = 10$ på träningsdatan.

För FNN varierades både antalet lager samt antalet noder i lagrena. I CNN-modellerna hölls den faltande arkitekturen konstant och fullt anslutna delen varierades. Den faltande arkitekturen bestod av två faltande lager som använde max-pooling. Gemensamt för samtliga modeller är att de använde ReLu som aktiveringsfunktion i lagrena, L_2-regularisering tillämpades inte samt att antalet epoker alltid var 10. Se appendix B för kod av modellerna samt appendix A för ytterligare beskrivning av nyckelorden. I tabell 4 redovisas noggrannheten för samtliga modeller vid korsvalidering samt modellernas arkitektur.

5 Resultat

I denna sektion framför vi resultaten av våra metoder. Vi konstaterar vad vi har fått för resultat och vi tar upp intressanta observationer. En diskussion om resultaten följer under rubriken diskussion.

5.1 Klassificering med medelvärde

När vi applicerade klassificering med medelvärde fick vi mellan 60 och 70 procent noggrannhet, för varje klass, vilket är ett acceptabelt resultat för en simpel metod. Den genomsnittliga noggrannheten var 68,6 procent. Detta innebär att vi förväntar oss att resterande metoder kommer ha en högre noggrannhet än 68,6 procent.

5.2 Klassificering med principalkomponenter

I tabell 2 visas resultaten för klassificering med bästa funktionella och bästa vektor approximationen. Notera att vi har en topp i noggrannhet runt 20 principalkomponenter för FPCA och för PCA är toppen mellan 60-90 principalkomponenter. Överlag verkar PCA ha bättre noggrannhet med en topp på 86,6% medan FPCA har en topp på 81,2%.

Tabell 3 visar istället resultaten för klassificering med ordinära respektive klassvisa principalkomponenter som har förts in i ett neuronnät. Här ser det ut som att FPCA och PCA har likartad noggrannhet. Vi ser att noggrannheten, på testdatan, varierar mindre när vi applicerar korsvalidering. Vi kan däremot inte urskilja någon tydlig topp i noggrannheten på dessa resultat men vi ser att noggrannheten verkar sjunka, på testdatan för FPCA-KV, om vi har mer principalkomponenter än fyrtio.

Tabell 2: Noggrannhet på träningsdatan av Fashion-MNIST med metoden bästa funktionella approximation och bästa vektorapproximation.

Metod	Antal principalkomponenter											
	5	10	20	30	60	90	120	150	180	250	350	500
FPCA	79,6	80,3	81,2	80,5	75,7	72,3	68,5	64,0	57,9	-	-	-
PCA	82,0	83,0	84,9	85,5	86,6	86,0	85,8	85,6	84,9	82,8	78,5	69,3

Tabell 3: Genomsnittlig noggrannhet på Fashion-MNIST för metoderna med principalkomponenter och neuronnät. Korsvalidering applicerades med $K = 10$. Suffixen -H och -KV anger huruvida dekompositionen utfördes på hela träningsdatan respektive klassvis för träningsdata. Kolumnen PK indikerar antal principalkomponenter som användes. Kolumnerna Träning och Test anger andelen korrekta klassifikationer i procent. Testdatan utgörs av den åsidosatta datan vid korsvalidering, se avsnitt 2.4.

PK	PCA-H		FPCA-H		PCA-KV		FPCA-KV	
	Träning	Test	Träning	Test	Träning	Test	Träning	Test
5	73,9	73,9	73,8	73,9	83,1	83,0	82,8	82,6
10	81,0	80,7	81,1	80,7	84,3	84,2	84,3	84,2
20	84,6	84,6	84,6	84,4	85,3	85,0	85,2	84,8
30	85,6	85,5	85,4	85,3	85,6	86,0	85,6	85,1
40	86,3	86,2	86,1	85,8	85,9	85,7	85,6	85,5
60	86,8	86,5	86,7	86,3	86,0	85,6	85,5	85,4
80	87,1	86,8	86,7	86,4	85,9	85,7	85,4	85,0

5.3 Klassificering med neuronnät

I tabell 4 har vi samlat prestandan för de olika nätverken på tränings- och testdata. Som väntat har FNN-modellerna lägre prestanda överlag än CNN, eftersom CNN är specialiserat på bilder. Vi ser också att prestationen verkar variera mer för FNN än för CNN. FNN modell 4 och CNN modell 5 är de modeller som presterar bäst på testdatan. De är 88,6 respektive 96,7 procent pricksäkra på testdatan.

Tabell 4: Genomsnittlig noggrannhet för neurala nätverk på Fashion-MNIST vid korsvalidering med K = 10. Testdatan utgörs av den åsidosatta datan vid korsvalidering, se avsnitt 2.4

FNN	**Arkitektur**	**Träning**	**Test**	**CNN**	**Arkitektur**	**Träning**	**Test**
Modell 1	[784,128,10]	85,0	84,2	Modell 1	[-,128,10]	98,1	96,2
Modell 2	[784,88,40,10]	88,8	88,4	Modell 2	[-,50,10]	97,6	95,8
Modell 3	[784,44,44,40,10]	89,0	88,4	Modell 3	[-,300,10]	98,3	96,4
Modell 4	[784,22,22,22,22,20,20,10]	89,3	88,6	Modell 4	[-,450,10]	98,4	96,3
Modell 5	[784,60,28,40,10]	88,9	87,8	Modell 5	[-,512,10]	98,5	96,7
Modell 6	[784,70,9,9,40,10]	86,5	85,8	Modell 6	[-,64,64,10]	97,8	95,9

5.4 Jämförelse av samtliga klassficieringsmetoder med fullständig träningsdata

Tabell 5 innehåller de bästa metodernas noggrannhet när de tränats på hela träningsdatan samt utvärderats på den hittills åsidosatta testdatan. Vid val av antal principalkomponenter för metoderna i tabell 3 har hänsyn tagits till hur komplexa modellerna är samt den avtagande förbättringen i noggrannhet.

Tabell 5: Jämförelse av samtliga klassificeringsmetoders noggrannhet på Fashion-MNIST med hela träningsdatan samt testdatan.

Klassifikationsmetod	**Noggrannhet (%)**	
	Träning	Test
Närmast medelvärde	68,6	67,7
Bästa vektorapproximation (60 PK)	86,6	83,6
Bästa funktionella approximation (20 PK)	81,2	80,4
PCA-H och NN (30 PK)	84,7	84,0
FPCA-H och NN (30 PK)	84,5	83,1
PCA-KV och NN (30 PK)	85,1	84,2
FPCA-KV och NN (40 PK)	85,1	82,9
FNN (Modell 4)	85,8	84,1
CNN (Modell 5)	98,1	90,3

6 Diskussion

Under denna sektion diskuteras resultaten av de olika metoderna som redovisades i sektion 5. Dessutom diskuteras det begränsade datasetets inverkan samt förslag till vidare studier framläggs.

6.1 Klassificering med medelvärde

Noggrannheten för klassificering med medelvärde blev ungefär 68 % på testdatan, se tabell 5. Med tanke på metodens simplicitet tycker vi att detta är ett ganska imponerande resultat. Metoden kräver trots allt ingen förståelse om principalkomponentanalys eller neuronnät, som kan uppfattas ganska komplexa, vilket övriga metoder baseras på. Dock kan det begränsade datasetet Fashion-MNIST varit fördelaktigt för denna metod, då bilderna är centrerade och roterade åt samma håll.

När vi klassificerar med medelvärde så undersöker vi bara hur bra vår metod klassificerar med euklidisk norm. Däremot kan det vara intressant att undersöka hur andra normer presterar i jämförelse och om en annan norm skulle kunna ge en högre noggrannhet än euklidisk norm.

6.2 Bästa vektorapproximation och funktionella approximation

De två olika metoderna bästa vektor- samt bästa funktionella approximation var snarlika till utförande. Tabell 5 visar att vektorapproximationen gav bättre resultat än den funktionella, vilket är rimligt då den senare approximerade datapunkterna till en spline med färre noder (219) jämfört med datapunkterna ($28^2 = 784$). I detta steg förloras sannolikt en del information som leder till den något försämrade noggrannheten, $\sim$3 % på testdatan. Däremot finns det ett syfte med den funktionella approximationen som kan bli tydligare för bilder med högre upplösning, vilket är att dimensionen av funktionsrummet är lägre än antalet pixlar i bilderna. Därmed utför FPCA beräkningar med mindre matriser. I ett sådant scenario kanske det är fördelaktigt att inte representera bilderna fullständigt för att få enklare beräkningar.

Fortsättningsvis visar sig ett mönster bland resultaten från bästa vektorapproximation och funktionella approximation, nämligen att det verkar finnas en gräns för när fler principalkomponenter inte längre förbättrar noggrannheten, utan till och med ger sämre resultat, se tabell 2. Det verkar alltså finnas en poäng med att bara betrakta de dimensioner som ger en representativ bild av datan och att betrakta övriga dimensioner som brus.

6.3 Klassificering med principalkomponenter och neuronnät

I denna sektion diskuteras resultaten i tabell 3. Tabellen visar att resultaten blir bättre med fler principalkomponenter. Detta verkar gälla upp till en gräns när noggrannheten inte längre förbättras avsevärt, utan bara ökar metodens komplexitet. Denna gräns kan väljas på olika sätt och vi har valt gränsen som det antalet principalkomponenter som står inom parentes i tabell 5.

Med väldigt få principalkomponenter verkar dom klassvisa principalkomponterna fungera bättre än dom ordinära att klassificera med. Dock är detta inte en helt rättvis jämförelse, på grund av att det totala antalet principalkomponenter är fler för det klassvisa fallet. För exemplet med 5 st, har 5 principalkomponenter använts från varje klass. Detta ger totalt 50 principalkomponenter och därmed större neuronnät i dom klassvisa metoderna men endast 5 principalkomponenter och mindre neuronnät i dom ordinära fallen. Detta gäller för alla antal principalkomponenter i tabellen men skillnaden blir märkbarast för små antal. Dessutom kan man föreställa sig andra klassificeringsscenarion där antalet klasser är större. Framtagningen av de klassvisa principalkomponenterna blir då mer mödosam samtidigt som skillnaden i storlek på neuronnäten i klassvisa/ordinära fallen då växer ytterligare. I ett sådant scenario skulle det kunna vara fördelaktigt att använda ordinär principalkomponentanalys istället för klassvis, men det återstår att studera.

Vidare, om vi jämför resultaten från metoderna principalkomponeter och neuronnät mot bästa vektor-/funktionella approximation i tabell 5, ser vi övergripande att de förstnämnda verkar prestera aningen bättre på testdatan. Däremot är metoderna som använder både principalkomponenter och neuronnät svårare att förstå samt implementera jämfört med bästa vektor-/funktionella approximation, eftersom att det kräver förståelse för både principalkomponenterna och neuronnät.

6.4 FNN och CNN

Från tabell 4 ser man att CNN modellerna har högre noggrannhet jämfört med FNN detta kan ses som rimligt då CNN har en struktur där näten i lagrena går igenom indatan i delar istället för hela indatan direkt. Den högre noggrannheten kan motiveras med att CNN strukturen låter lagrena leta efter mindre strukturer viktiga för objekten som exempelvis kanter. CNN har också bästa resultaten av alla metoder undersökta i rapporten.

Det bör nämnas att båda metoderna tycktes ha generellt bättre noggrannhet efter steg i korsvalideringen detta kan tyda på att något är fel i detta fall är en trolig orsak att nätvärken inte återställs som det ska mellan steg av korsvalideringen.

Det borde också noteras att CNN hade störst skillnad mellan noggrannheten på tränings- och testdatan. Detta kan tyda på att vi tränat nätet för mycket och att det således memorerat datan istället för att identifiera generella egenskaper, så kallat överanpassning.

6.5 Styrkor och svagheter för de olika metoderna

I tabell 5 ser vi att alla metoderna har relativt liknade resultat bortset från medelvärdes klassificering vilket kan ses som rimligt då den är enklaste metoden.

Två likande metoder är bästa vektorapproximatio och bästa funktionella approximation utav dessa presterade bästa vektorapproximatio bättre (funktionella approximation hade lägst noggrannhet förutom medelvärdes klassificering). Det bör också nämnas att funktionella approximationen verkade vara relativt känslig för valet antalet principalkomponenter (PK) jämfört med vektorapproximatio. Båda hade dock tyckesvis sämre noggrannhet vid större antal PK.

Artificiella nätverks metoderna hade generellt bra resultat (några dock lägre noggrannhet än bästa vektorapproximationen). CNN hade högst noggrannhet på både tränings- och testdata. Noggrannheten var kring 5 procentenheter högre än de andra metoderna. Detta kan dock vara specifikt för datan då det skulle kunna tänkas att metoderna med dimensionsminskning får förhållandevis bättre noggrannhet för data med hög dimension.

6.6 Begränsningar med datan

Eftersom datauppsättning endast innehåller kläder är det inte säkert att metoderna behandlade i denna rapport kan appliceras på andra datauppsättningar med samma resultat.

Ett viktigt karaktärsdrag hos datan är att klädesplaggen presenteras på ett standardiserat sätt och är därför relativt enkla objekt. Detta kan innebära att metoderna ger mycket andra resultat

på objekt som presentersas på slumpartade sätt exempelvis i en hög eller upp och ner. Slumpartad data hade troligt varit extra besvärligt för PCA och FPCA metoderna då dessa utnyttjar standardiseringen för att minska dimensionen på datan.

Det andra troligen viktiga karaktärsdraget är att objekten (klädesplaggen) är ensamma i bilden. Data med bilder som innehåller ett flertal objekt en del av intresse och en del inte skulle troligvis innebära att användingen av metoderna som presenterats i denna rapport hade behövts omarbetas kraftigt.

6.7 Förslag till framtida studier

I denna del presenteras utvecklingsmöjligheter hos de FPCA/PCA-baserade metoderna samt ytterligare parametrar att analysera för vidare studier.

6.7.1 Förbättringsområden

Placeringen av noderna som användes för spline-funktionerna kan utvecklas. I detta arbete placerades noderna uniformt över intervallet $[1, 28^2]$. Genom att placera noderna tätare där datan varierar kraftigt, kring konturen hos de olika bilderna, och glesare där datan varierar mindre, generellt vid hörn och kanter, skulle spline-representationerna återspegla originaldatan bättre. Detta på grund av att en spline med få noder har svårt att väl approximera en funktion som varierar mycket.

Vidare behandlades bilderna som endimensionella vektorer, och ingen hänsyn tas då till närliggande pixlar i y-led. Metoden med CNN är den enda som tar hänsyn till de två dimensionerna på något sätt, genom faltningen. Ett sätt att använda datans information i y-led, som troligen är ganska komplicerat, hade varit att försöka anpassa tvådimensionella funktioner till datan.

6.7.2 Fler parametrar att jämföra hos de olika metoderna

Vid jämförelse av olika klassificeringsmetoder finns det många aspekter att beakta och det kan därför vara svårt att utse en metod till den bästa. I denna studie undersöktes främst noggrannheten hos de olika metoderna, och resultatet blev att klassificering med CNN utan PCA presterade bäst med $\sim 90\%$ noggrannhet. Dock undersöktes inte exempelvis hur lång tid algoritmerna tar eller hur mycket minneskapacitet de olika metoderna kräver. Ett syfte med PCA är att reducera antalet dimensioner hos datan. Detta skulle möjligtvis kunna minska antalet beräkningar och lagringsutrymmet för PCA-baserade metoder, och för vissa tillämpningar kanske detta är att föredra. FPCA tar dessutom detta ett steg längre, genom att innan PCA approximera datan med funktioner.

7 Slutsats

Resultaten visade att CNN var den metod med bäst noggrannhet (90,4 %). Detta var inte helt oväntat då CNN:s är effektiva och vanligt förekommande inom bildigenkänningsalgoritmer. Vidare såg vi att metoderna bästa vektor-/funktionella approximation, samt de fyra metoderna som använde principalkomponenter och neuronnät, hade ungefär lika hög noggrannhet (80-84 %) samt nådde upp till FNN:s noggrannhet (84 %). Fler principalkomponenter gav i regel högre noggrannhet upp till en gräns där noggrannheten stagnerade eller började avta. I denna studie har främst noggrannheten undersökts men vid jämförelse av flera parametrar, såsom beräkningstid och minneskapacitet, finns det möjligheter att de FPCA-baserade metoderna kan ha andra fördelar.

Referenser

[1] Forsyth, D. A. och Ponce, J. *Computer vision: a modern approach*. Vol. 1. Pearson, 2012, s. xviii.

[2] Krizhevsky, A., Sutskever, I. och Hinton, G. E. "Imagenet classification with deep convolutional neural networks". *Advances in neural information processing systems* 25 (2012), s. 1097–1105.

[3] Russakovsky, O. m. fl. "ImageNet Large Scale Visual Recognition Challenge". *International Journal of Computer Vision (IJCV)* 115.3 (2015), s. 211–252. DOI: 10.1007/s11263-015-0816-y.

[4] McCulloch, W. S. och Pitts, W. "A logical calculus of the ideas immanent in nervous activity". *The bulletin of mathematical biophysics* 5.4 (1943), s. 115–133.

[5] Saha, S. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (hämtad 2021-04-02).

[6] Ramsay, J. och Silverman, B. *Functional Data Analysis*. 2. utg. Springer, 2005, s. 149–152.

[7] Chen, L. "Curse of Dimensionality". I: *Encyclopedia of Database Systems*. Utg. av L. LIU och M. T. ÖZSU. Boston, MA: Springer US, 2009, s. 545–546. DOI: 10.1007/978-0-387-39940-9_133.

[8] Rawat, W. och Wang, Z. "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review". *Neural Computation* 29.9 (2017), s. 2352–2449. URL: https://doi.org/10.1162/neco_a_00990 (hämtad 2021-02-01).

[9] Xijia, L. och Krzysztof, P. *Splinets: Functional Data Analysis using Splines and Orthogonal Spline Bases*. https://CRAN.R-project.org/package=Splinets. 2021. (Hämtad 2021-02-25).

[10] Chollet, F. *Keras.io*. 2005. URL: https://keras.io/api/#keras-api-reference (hämtad 2021-04-01).

[11] Google. *Google Colaboratory*. URL: https://colab.research.google.com (hämtad 2021-05-11).

[12] Holmåker, K. och Gustafsson, I. *Linjär algebra: fortsättningskurs*. 1. utg. Liber, 2016, s. 1–20. ISBN: 9789147112456.

[13] Young, N. *An Introduction to Hilbert Space*. 1. utg. Camebridge University Press, 1988, s. 21–28. ISBN: 9780521330718. DOI: 10.1017/CBO9781139172011.

[14] Conway, J. B. *A Course in Functional Analysis*. 2. utg. Graduate Texts in Mathematics ; 96. Springer Science+Business Media, 1990, s. 7–11. ISBN: 9781441930927.

[15] Podgórski, K. *Splinets – splines through the Taylor expansion, their support sets and orthogonal bases*. 2021. arXiv: 2102.00733 [stat.CO]. URL: https://arxiv.org/abs/2102.00733 (hämtad 2021-02-18).

[16] James, G. m. fl. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014, s. 30, 374–377. ISBN: 1461471370.

[17] Bishop, C. M. *Pattern recognition and machine learning*. Springer, 2006.

[18] Ramsundar, B. och Bosagh Zadeh, R. I: *TensorFlow for Deep Learning*. O'Reilly Media, Inc, mars 2018. ISBN: 9781491980453.

[19] Mahajan, P. *Fully Connected vs Convolutional Neural Networks*. https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5. 2020. (Hämtad 2021-04-05).

[20] Hastie, T., Tibshirani, R. och Friedman, J. *The elements of statistical learning: data mining, inference, and prediction*. 2. utg. Springer Science, 2009.

[21] Xiao, H., Rasul, K. och Vollgraf, R. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 28 aug. 2017. arXiv: cs.LG/1708.07747 [cs.LG]. URL: https://arxiv.org/abs/1708.07747 (hämtad 2021-02-05).

[22] Yegulalp, S. *What is TensorFlow? The machine learning library explained*. 2019. URL: https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html (hämtad 2021-04-01).

[23] Chollet, F. *Keras API reference: Fit-method*. Keras. URL: https://keras.io/api/models/model_training_apis/#fit-method (hämtad 2021-05-12).

[24] Chollet, F. *Keras API reference: ReLU layer*. Keras. URL: https://keras.io/api/layers/activation_layers/relu/ (hämtad 2021-05-12).

[25] Chollet, F. *Keras API reference: Softmax layer*. Keras. URL: https://keras.io/api/layers/activation_layers/softmax/ (hämtad 2021-05-12).
[26] Chollet, F. *Keras API reference: Adam*. Keras. URL: https://keras.io/api/optimizers/adam/ (hämtad 2021-05-12).
[27] Chollet, F. *Keras API reference: Accuracy metrics*. Keras. URL: https://keras.io/api/metrics/accuracy_metrics/ (hämtad 2021-05-12).
[28] Chollet, F. *Keras API reference: SparseCategoricalCrossentropy*. Keras. URL: https://keras.io/api/losses/probabilistic_losses/#sparse_categorical_crossentropy-function (hämtad 2021-05-12).

A Ordlista

Tensorflow
Tensorflow är ett bibliotek med öppen källkod för numerisk beräkning och storskalig maskininlärning. Tensorflow använder Python för att tillhandahålla ett bekvämt frontend-API för att bygga applikationer med ramverket.[22]

Keras
Keras är ett djupinlärnings-API skrivet i Python som körs ovanpå maskininlärningsplattformen TensorFlow. Den utvecklades med fokus på att möjliggöra snabba experiment.[10]

Epoch
Epoch (Epok på svenska) är en hyperparameter som definierar antalet gånger som inlärningsalgoritmen ska iterera genom hela träningsdataset. En epok innebär att varje prov i träningsdatasetet har haft möjlighet att uppdatera de interna modellparametrarna.[23]

ReLU
ReLU är en förkortning av *Rectified Linear Unit* och är en aktiveringsfunktion tillgänglig i Keras. Funktionen ReLU ges som $f(x) =$ `max(x, 0)` med sina standardvärden. Det finns andra aktiveringsfunktioner såsom sigmoid, softplus, softmax och tanh.[24]

Softmax
Softmax omvandlar en vektor med reella värden till en vektor med kategoriska sannolikheter. Kategoriska sannolikheter innebär att varje element i utgångsvektorn har värden i intervallet [0,1] och summan av samtliga element är 1. Softmax används ofta i sista lagret för att utvärdet i neuronnätet skall standardiseras i intervallet [0,1]. Utvärdet av softmax kan tolkas som en sannolikhetsfördelning. Standardfunktionen för softmax ges av $f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$ där $x \in \mathbb{R}^N$. [25].

Adam
Adam är optimeringsalgoritm som finns implementerad i Keras. Det är en stokastisk nedstigningsmetod som bygger på adaptiv uppskattning av första ordningens och andra ordningens matematiska moment.[26]

Accuracy
Måttet *Accuracy* i Keras är ekvivalent med begreppet noggrannhet, det vill säga andelen korrekta klassifikationer från samtliga klassifikationer. [27]

Sparse-Categorical-Crossentropy
Sparse-Categorical-Crossentropy är en förlustfunktion (objektivfunktion) i Keras som ofta används vid kategorisk klassificering. [28]

B Modeller för neurala nätverk

```
FNN_modell_1=keras.Sequential([
                        keras.layers.Flatten(input_shape=(28,28)),
                        keras.layers.Dense(128, activation='relu'),
                        keras.layers.Dense(10, activation='softmax')
])
```

```
FNN_modell_2=keras.Sequential([
                         keras.layers.Flatten(input_shape=(28,28)),
                         keras.layers.Dense(88, activation='relu'),
                         keras.layers.Dense(40, activation='relu'),
                         keras.layers.Dense(10, activation='softmax'),
])
```

```
FNN_modell_3=keras.Sequential([
                         keras.layers.Flatten(input_shape=(28,28)),
                         keras.layers.Dense(44, activation='relu'),
                         keras.layers.Dense(44, activation='relu'),
                         keras.layers.Dense(40, activation='relu'),
```

```
                            keras.layers.Dense(10, activation='softmax'),
])
```

```
FNN_modell_4=keras.Sequential([
                            keras.layers.Flatten(input_shape=(28,28)),
                            keras.layers.Dense(22, activation='relu'),
                            keras.layers.Dense(22, activation='relu'),
                            keras.layers.Dense(22, activation='relu'),
                            keras.layers.Dense(22, activation='relu'),
                            keras.layers.Dense(20, activation='relu'),
                            keras.layers.Dense(20, activation='relu'),
                            keras.layers.Dense(10, activation='softmax'),
])
```

```
FNN_modell_5=keras.Sequential([
                            keras.layers.Flatten(input_shape=(28,28)),
                            keras.layers.Dense(60, activation='relu'),
                            keras.layers.Dense(28, activation='relu'),
                            keras.layers.Dense(40, activation='relu'),
                            keras.layers.Dense(10, activation='softmax'),
])
```

```
FNN_modell_6=keras.Sequential([
                            keras.layers.Flatten(input_shape=(28,28)),
                            keras.layers.Dense(70, activation='relu'),
                            keras.layers.Dense(9, activation='relu'),
                            keras.layers.Dense(9, activation='relu'),
                            keras.layers.Dense(40, activation='relu'),
                            keras.layers.Dense(10, activation='softmax'),
])
```

```
CNN_modell_1 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
])
```

```
CNN_modell_2 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(50, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
])
```

```
CNN_modell_3 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(300, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
])
```

```
CNN_modell_4 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
```

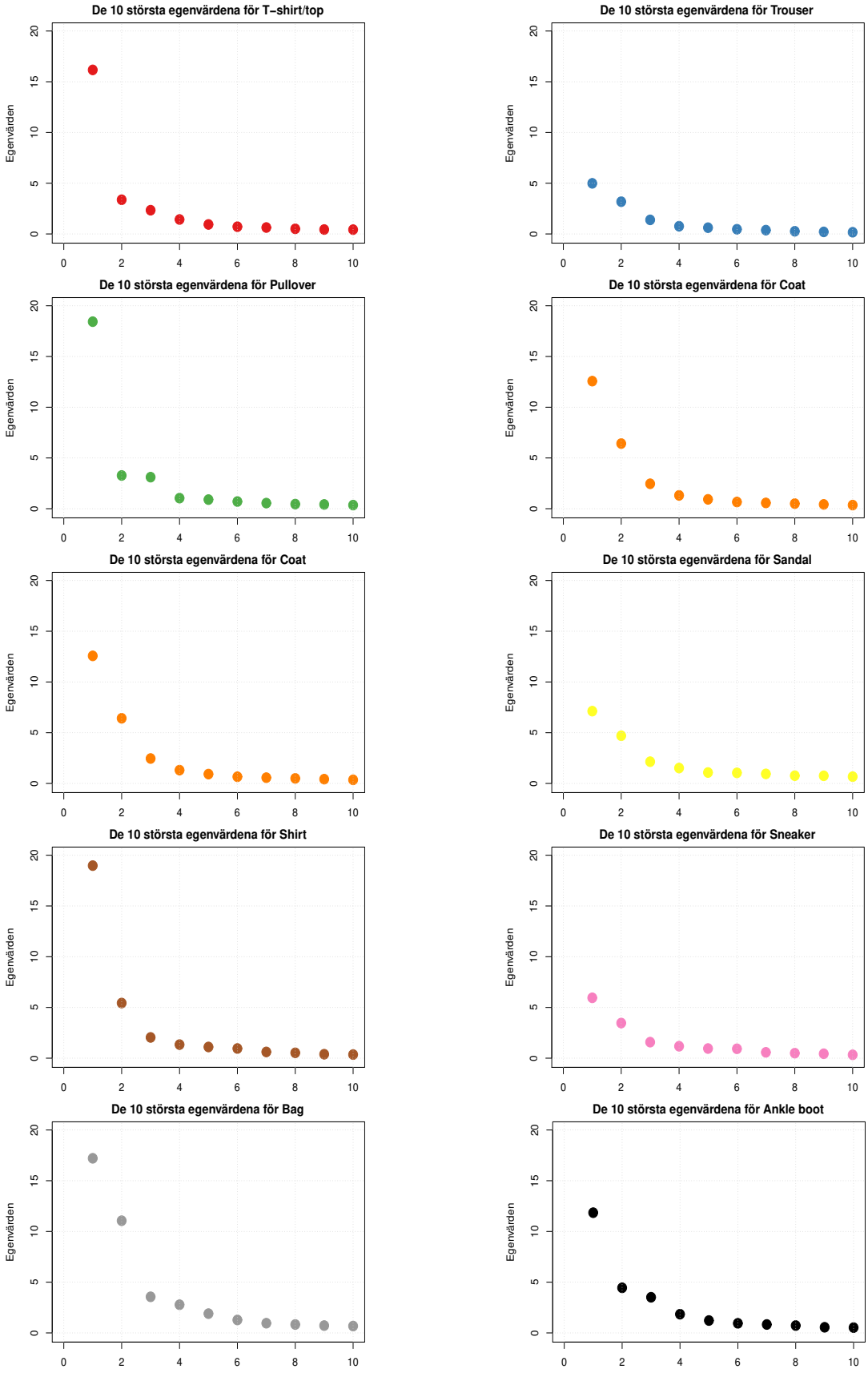
```
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(450, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
])
```

```
CNN_modell_5 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
])
```

```
CNN_modell_6 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation=tf.nn.relu),
    tf.keras.layers.Dense(64, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
])
```

C Plots av egenvärden för de olika klasserna

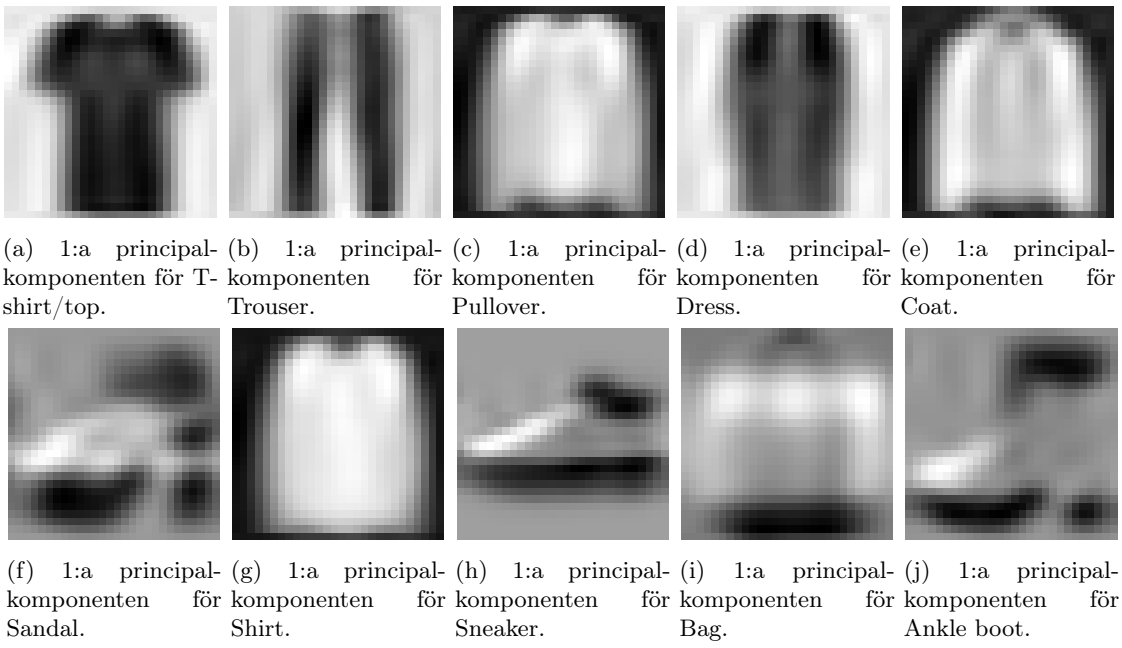
Här presenteras de 10 största egenvärdena, $\lambda_C^{k,1}, ..., \lambda_C^{k,10}$, för respektive klass i separata figurer.



Figur 13: Sammanställning av egenvärdena $\lambda_C^{k,1}, ..., \lambda_C^{k,10}$ för respektive klass

D Bilden av första principalkomponenten för respektive klass

Här presenteras bilderna som genereras från första principalkomponenten för respektive klass.



(a) 1:a principalkomponenten för T-shirt/top.

(b) 1:a principalkomponenten för Trouser.

(c) 1:a principalkomponenten för Pullover.

(d) 1:a principalkomponenten för Dress.

(e) 1:a principalkomponenten för Coat.

(f) 1:a principalkomponenten för Sandal.

(g) 1:a principalkomponenten för Shirt.

(h) 1:a principalkomponenten för Sneaker.

(i) 1:a principalkomponenten för Bag.

(j) 1:a principalkomponenten för Ankle boot.

Figur 14: Sammanställning av bilderna av de första principalkomponenterna för samtliga klasser. Funktionerna är normerade.

E Kod

Nedan redovisas koden som skrivits i detta arbete. Koden från R-filerna samt koden som skrivits i Colab finns tillgänglig via ett repository på GitHub här: `https://github.com/hataloo/FPCAvsNN`.

E.1 Import av data i R, allmän databehandling samt omvandling till Splines

```
library(Splinets)
library(keras)
library(pbapply)
library(tictoc)
library(pracma)
library(RcppCNPy)

fashion_mnist <- dataset_fashion_mnist()

c(train_images, train_labels) %<-% fashion_mnist$train
c(test_images, test_labels) %<-% fashion_mnist$test
class_names = c('T-shirt/top',
                'Trouser',
                'Pullover',
                'Dress',
                'Coat',
                'Sandal',
                'Shirt',
                'Sneaker',
                'Bag',
                'Ankle boot')
number_of_images <- dim(train_labels)
image_size <- dim(train_images)[2]
order <- 4
number_of_knots <- image_size*8
knots <- seq(from = 1, to = image_size^2, length.out = number_of_knots)


flipped_train_images <- array(0,dim = c(number_of_images, image_size, image_size))
flipped_test_images <- array(0, dim = c(length(test_labels), image_size,
    image_size))
for(i in 1:number_of_images){
  flipped_train_images[i,,] <- t(apply(train_images[i,,],2,rev))/255
}
for(i in 1:length(test_labels)){
  flipped_test_images[i,,] <- t(apply(test_images[i,,],2,rev))/255
}
train_vectors <- flipped_train_images
dim(train_vectors) <- c(number_of_images, image_size^2)
test_vectors <- flipped_test_images
dim(test_vectors) <- c(length(test_labels), image_size^2)

class_means <- array(dim = c(length(class_names), image_size^2))
tic("Calculating and projecting means")
for (class_index in 1:length(class_names)){
  class_means[class_index,] <- colMeans(train_vectors[train_labels == class_index
    -1, ])
}
class_mean_splines <- project(cbind(seq(image_size^2), t(class_means)), knots,
    order)
toc()
#To check if mean is similar.
#image(1:28, 1:28, matrix(class_means[3,], ncol = 28, nrow = 28, byrow = FALSE),
    col = gray((0:255/(255))), xaxt = "n", yaxt = "n")
#image(1:28, 1:28, matrix(evspline(class_mean_splines$sp, x = seq(28^2))[,10],
    ncol = 28, nrow = 28, byrow = FALSE), col = gray((0:255/(255))), xaxt = "n",
    yaxt = "n")

tic("Projecting the data onto Splines")
train_vector_splines <- project(cbind(seq(image_size^2), t(train_vectors)), knots,
     order)
```

```
toc()
test_vector_splines <- project(cbind(seq(image_size^2), t(test_vectors)), knots,
    order)

npySave("principalcoeffdata/train_labels.npy", train_labels)
npySave("principalcoeffdata/test_labels.npy", test_labels)
```

E.2 Utförande av ordinär funktionell principalkomponentanalys

```
library(Splinets)
library(pbapply)
library(tictoc)
library(pracma)

###FPCA centered around entire dataset
number_of_components_to_save <- min(800, number_of_knots - order -1)
principal_coeffs_matrix_all <- array(dim = c(number_of_images,
    number_of_components_to_save))
normalized_principal_coeffs_matrix_all <- array(dim = c(number_of_images,
    number_of_components_to_save))
test_principal_coeffs_matrix_all <- array(dim = c(length(test_labels),
    number_of_components_to_save))
test_normalized_principal_coeffs_matrix_all <- array(dim = c(length(test_labels),
    number_of_components_to_save))



dataset_mean_spline_coeffs <- colMeans(train_vector_splines$coeff)
sigma_all <- cov(sweep(train_vector_splines$coeff,2,dataset_mean_spline_coeffs))
spect_all <- eigen(sigma_all, symmetric = T)

principal_class_sp_all <- lincomb(train_vector_splines$basis, t(spect_all$vectors)
    )
principal_coeffs_matrix_all <- ((sweep(train_vector_splines$coeff,2,
    dataset_mean_spline_coeffs))
                                %*% spect_all$vectors)[,1:
    number_of_components_to_save]
for(component_index in 1:number_of_components_to_save){
  normalized_principal_coeffs_matrix_all[,component_index] <-
    principal_coeffs_matrix_all[,component_index]/sqrt(spect_all$values[
    component_index])
}
test_principal_coeffs_matrix_all <- ((sweep(test_vector_splines$coeff,2,
    dataset_mean_spline_coeffs))
                                %*% spect_all$vectors)[,1:
    number_of_components_to_save]
for(component_index in 1:number_of_components_to_save){
  test_normalized_principal_coeffs_matrix_all[,component_index] <-
    test_principal_coeffs_matrix_all[,component_index]/sqrt(spect_all$values[
    component_index])
}


npySave("principalcoeffdata/principal_coeffs_matrix_entire_dataset.npy",
    principal_coeffs_matrix_all)
npySave("principalcoeffdata/test_principal_coeffs_matrix_entire_dataset.npy",
    test_principal_coeffs_matrix_all)
npySave("principalcoeffdata/normalized_principal_coeffs_matrix_entire_dataset.npy"
    , normalized_principal_coeffs_matrix_all)
npySave("principalcoeffdata/test_normalized_principal_coeffs_matrix_entire_dataset
    .npy", test_normalized_principal_coeffs_matrix_all)
```

E.3 Utförande av ordinär principalkomponentanalys

```
##Need to run data_preparation before.

###PCA centered around entire dataset

number_of_components_to_save <- 100
```

```
pca_coeffs_matrix_all <- array(dim = c(number_of_images,
    number_of_components_to_save))
normalized_pca_coeffs_matrix_all <- array(dim = c(number_of_images,
    number_of_components_to_save))

test_pca_coeffs_matrix_all <- array(dim = c(length(test_labels),
    number_of_components_to_save))
test_normalized_pca_coeffs_matrix_all <- array(dim = c(length(test_labels),
    number_of_components_to_save))

dataset_mean <- colMeans(train_vectors)
pca_sigma_all <- cov(sweep(train_vectors,2, dataset_mean))
pca_spect_all <- eigen(pca_sigma_all, symmetric = T)

pca_coeffs_matrix_all <- ((sweep(train_vectors,2,dataset_mean)) %*% pca_spect_all$
    vectors)[,1:number_of_components_to_save]
test_pca_coeffs_matrix_all <- ((sweep(test_vectors,2,dataset_mean)) %*%
    pca_spect_all$vectors)[,1:number_of_components_to_save]
for(component_index in 1:number_of_components_to_save){
  normalized_pca_coeffs_matrix_all[,component_index] <- pca_coeffs_matrix_all[,
    component_index]/sqrt(pca_spect_all$values[component_index])
  test_normalized_pca_coeffs_matrix_all[,component_index] <-
    test_pca_coeffs_matrix_all[,component_index]/sqrt(pca_spect_all$values[
    component_index])
}




npySave("principalcoeffdata/pca_coeffs_matrix_entire_dataset.npy",
    pca_coeffs_matrix_all)
npySave("principalcoeffdata/test_pca_coeffs_matrix_entire_dataset.npy",
    test_pca_coeffs_matrix_all)
npySave("principalcoeffdata/normalized_pca_coeffs_matrix_entire_dataset.npy",
    normalized_pca_coeffs_matrix_all)
npySave("principalcoeffdata/test_normalized_pca_coeffs_matrix_entire_dataset.npy",
     test_normalized_pca_coeffs_matrix_all)
```

E.4 Utförande av klassvis funktionell principalkomponentanalys

```
library(Splinets)
library(pbapply)
library(tictoc)
library(pracma)


###FPCA centered around every class
#Consider lowering number_of_components_to_save to lower RAM-usage.
number_of_components_to_save <- min(800, number_of_knots - order -1)
principal_coeffs_array <- array(dim = c(number_of_images, length(class_names),
    number_of_components_to_save))
normalized_principal_coeffs_array <- array(dim = c(number_of_images, length(
    class_names), number_of_components_to_save))
test_principal_coeffs_array <- array(dim = c(length(test_labels), length(
    class_names), number_of_components_to_save))
test_normalized_principal_coeffs_array <- array(dim = c(length(test_labels),
    length(class_names), number_of_components_to_save))


sigma <- array(dim = c(length(class_names), number_of_knots - order -1,
    number_of_knots - order -1))
spect <- list()
principal_class_sp <- list()
for(projection_index in 1:length(class_names)){
  #centered_coeffs <- array(dim = c(sum(train_labels == projection_index-1),
    number_of_knots-order-1))
  #train_class_spline_coeffs <- (train_vector_splines$coeff[train_labels==(
    projection_index-1),])
  #for(sample_index in 1:(dim(centered_coeffs)[1])){centered_coeffs[sample_index,]
     <- train_class_spline_coeffs[sample_index,] - class_mean_splines$coeff[
```

```
    projection_index,] }
  #sigma[projection_index,,] <- cov(centered_coeffs)
  sigma[projection_index,,] <- cov(sweep(train_vector_splines$coeff[train_labels
    == (projection_index-1),],2,class_mean_splines$coeff[projection_index,]))
  #sigma[projection_index,,] <- cov(train_vector_splines$coeff[train_labels == (
    projection_index-1),] - class_mean_splines$coeff[projection_index,])
  spect[[projection_index]] <- eigen(sigma[projection_index,,], symmetric = TRUE)
}
pb <- timerProgressBar(min = 0, max = length(class_names))
for(projection_index in 1:length(class_names)){
  principal_class_sp[[projection_index]] <- lincomb(train_vector_splines$basis, t(
    spect[[projection_index]]$vectors))
  setTimerProgressBar(pb, value = projection_index)
}

for(projection_index in 1:length(class_names)){
  principal_coeffs_array[,projection_index,] <- (sweep(train_vector_splines$coeff,
     2, class_mean_splines$coeff[projection_index,]) %*% spect[[projection_index]]
    $vectors)[,1:number_of_components_to_save]
  test_principal_coeffs_array[,projection_index,] <- (sweep(test_vector_splines$
    coeff, 2, class_mean_splines$coeff[projection_index,]) %*% spect[[
    projection_index]]$vectors)[,1:number_of_components_to_save]
  for(component_index in 1:number_of_components_to_save){
    normalized_principal_coeffs_array[,projection_index,component_index] <-
    principal_coeffs_array[,projection_index, component_index]/sqrt((spect[[
    projection_index]]$values[component_index]))
    test_normalized_principal_coeffs_array[,projection_index,component_index] <-
    test_principal_coeffs_array[,projection_index, component_index]/sqrt((spect[[
    projection_index]]$values[component_index]))
  }
}

### Testing if the principal components look reasonable.
#Reasonable if they have some resemblance of the original class.
class_index <- 1
component_index <- 1
spline_eval <- evspline(principal_class_sp[[class_index]], sID = 1:5, seq(
    image_size^2))
for(i in 1:3){image(1:28, 1:28, matrix(spline_eval[,i+1], ncol = 28, nrow = 28,
    byrow = FALSE), col = gray((0:255/(255))), xaxt = "n", yaxt = "n")
}
mean_eval <- evspline(class_mean_splines$sp, sID = class_index, seq(image_size^2))
image(1:28, 1:28, matrix(spline_eval[,component_index+1]*spect[[class_index]]$
    values[component_index] + mean_eval[,2], ncol = 28, nrow = 28, byrow = FALSE),
     col = gray((0:255/(255))), xaxt = "n", yaxt = "n")

eig_funcs_to_use <- 5
spline_eval <- evspline(principal_class_sp[[class_index]], sID = 1:
    eig_funcs_to_use, seq(image_size^2))
mean_eval <- evspline(class_mean_splines$sp, sID = class_index, seq(image_size^2))
pixel_value <- array(mean_eval[,2], dim = c(image_size^2))
for(i in 1:eig_funcs_to_use){
  pixel_value <- pixel_value + principal_coeffs_array[sample_index,class_index,i]*
    spline_eval[,i+1]
}


#Verify that the coefficients have mean ~0 and var ~1,
#hist should hopefully resemble a Gaussian but not necessary.
class_index <- 1
component_index <- 1
hist(normalized_principal_coeffs_array[train_labels == class_index-1,class_index,
    component_index], breaks = 40)
mean(normalized_principal_coeffs_array[train_labels == class_index-1,class_index,
    component_index])
var(normalized_principal_coeffs_array[train_labels == class_index-1,class_index,
    component_index])

number_of_components_in_matrix <- 80
principal_coeffs_matrix <- matrix(principal_coeffs_array, nrow = number_of_images,
```

```
     ncol = length(class_names)*number_of_components_in_matrix)
normalized_principal_coeffs_matrix <- matrix(normalized_principal_coeffs_array,
    nrow = number_of_images, ncol = length(class_names)*
    number_of_components_in_matrix)
test_principal_coeffs_matrix <- matrix(test_principal_coeffs_array, nrow = length(
    test_labels), ncol = length(class_names)*number_of_components_in_matrix)
test_normalized_principal_coeffs_matrix <- matrix(
    test_normalized_principal_coeffs_array, nrow = length(test_labels), ncol =
    length(class_names)*number_of_components_in_matrix)


save(principal_coeffs_array, file = "principalcoeffdata/principal_coeffs_array.
    RData")
save(normalized_principal_coeffs_array, file = "principalcoeffdata/
    principal_coeffs_array.RData")
save(principal_class_sp, file = "principalcoeffdata/principal_class_sp.RData")
save(spect, file = "principalcoeffdata/spect_classwise.RData")
npySave("principalcoeffdata/principal_coeffs_matrix.npy", principal_coeffs_matrix)
npySave("principalcoeffdata/principal_coeffs_matrix_cut.npy",
    principal_coeffs_matrix[,1:(10*10)])
npySave("principalcoeffdata/normalized_principal_coeffs_matrix.npy",
    normalized_principal_coeffs_matrix)
npySave("principalcoeffdata/normalized_principal_coeffs_matrix_cut.npy",
    normalized_principal_coeffs_matrix[,1:(10*10)])


npySave("principalcoeffdata/test_principal_coeffs_matrix.npy",
    test_principal_coeffs_matrix)
npySave("principalcoeffdata/test_normalized_principal_coeffs_matrix.npy",
    test_normalized_principal_coeffs_matrix)

class_index <- 1
component_index <- 1
hist(normalized_principal_coeffs_matrix[train_labels == class_index-1,class_index
    ], breaks = 40)
mean(normalized_principal_coeffs_matrix[train_labels == class_index-1,class_index
    ])
var(normalized_principal_coeffs_matrix[train_labels == class_index-1,class_index])
```

E.5 Utförande av klassvis principalkomponentanalys

```
##Need to run data_preparation before.

###PCA centered around every class
number_of_components_to_save <- 80
pca_coeffs_array <- array(dim = c(number_of_images, length(class_names),
    number_of_components_to_save))
normalized_pca_coeffs_array <- array(dim = c(number_of_images, length(class_names)
    , number_of_components_to_save))
test_pca_coeffs_array <- array(dim = c(length(test_labels), length(class_names),
    number_of_components_to_save))
test_normalized_pca_coeffs_array <- array(dim = c(length(test_labels), length(
    class_names), number_of_components_to_save))

pca_sigma_cw <- array(dim = c(length(class_names), image_size^2, image_size^2))
pca_spect_cw <- list()
pb <- timerProgressBar(min = 0, max = length(class_names))
for(projection_index in 1:length(class_names)){
  pca_sigma_cw[projection_index,,] <- cov(sweep(train_vectors[train_labels == (
    projection_index-1),],2,class_means[projection_index,]))
  pca_spect_cw[[projection_index]] <- eigen(pca_sigma_cw[projection_index,,],
    symmetric = TRUE)
  setTimerProgressBar(pb, value = projection_index)
}
pb <- timerProgressBar(min = 0, max = length(class_names))
for(projection_index in 1:length(class_names)){
  pca_coeffs_array[,projection_index,] <- (sweep(train_vectors, 2, class_means[
    projection_index,]) %*% pca_spect_cw[[projection_index]]$vectors)[,1:
    number_of_components_to_save]
  test_pca_coeffs_array[,projection_index,] <- (sweep(test_vectors, 2, class_means
```

```
    [projection_index,]) %*% pca_spect_cw[[projection_index]]$vectors)[,1:
    number_of_components_to_save]
  for(component_index in 1:number_of_components_to_save){
    normalized_pca_coeffs_array[,projection_index,component_index] <-
    pca_coeffs_array[,projection_index, component_index]/sqrt(pca_spect_cw[[
    projection_index]]$values[component_index])
    test_normalized_pca_coeffs_array[,projection_index,component_index] <-
    test_pca_coeffs_array[,projection_index, component_index]/sqrt(pca_spect_cw[[
    projection_index]]$values[component_index])
  }
  setTimerProgressBar(pb, value = projection_index)
}

#Verify that the coefficients have mean ~0 and var ~1,
#hist should hopefully resemble a Gaussian but not necessary.
class_index <- 10
component_index <- 1
hist(normalized_pca_coeffs_array[train_labels == (class_index-1),class_index,
    component_index], breaks = 100)
mean(normalized_pca_coeffs_array[train_labels == (class_index-1),class_index,
    component_index])
var(normalized_pca_coeffs_array[train_labels == (class_index-1),class_index,
    component_index])

image(1:28, 1:28, matrix(pca_spect_cw[[8]]$vectors[,1], ncol = 28, nrow = 28,
    byrow = FALSE), col = gray((0:255/(255))), xaxt = "n", yaxt = "n")
image(1:28, 1:28, matrix(class_means[6,], ncol = 28, nrow = 28, byrow = FALSE),
    col = gray((0:255/(255))), xaxt = "n", yaxt = "n")
image(1:28, 1:28, matrix(colMeans(train_vectors[train_labels == 5,]), ncol = 28,
    nrow = 28, byrow = FALSE), col = gray((0:255/(255))), xaxt = "n", yaxt = "n")



number_of_components_in_matrix <- 80
principal_coeffs_matrix <- matrix(pca_coeffs_array, nrow = number_of_images, ncol
    = length(class_names)*number_of_components_in_matrix)
normalized_pca_coeffs_matrix <- matrix(normalized_pca_coeffs_array, nrow =
    number_of_images, ncol = length(class_names)*number_of_components_in_matrix)
test_pca_coeffs_matrix <- matrix(test_pca_coeffs_array, nrow = length(test_labels)
    , ncol = length(class_names)*number_of_components_in_matrix)
test_normalized_pca_coeffs_matrix <- matrix(test_normalized_pca_coeffs_array, nrow
     = length(test_labels), ncol = length(class_names)*
    number_of_components_in_matrix)



npySave("principalcoeffdata/pca_coeffs_matrix.npy", pca_coeffs_matrix_all)
npySave("principalcoeffdata/test_pca_coeffs_matrix.npy", test_pca_coeffs_matrix)
npySave("principalcoeffdata/normalized_pca_coeffs_matrix.npy",
    normalized_pca_coeffs_matrix)
npySave("principalcoeffdata/test_normalized_pca_coeffs_matrix.npy",
    test_normalized_pca_coeffs_matrix)
```

E.6 Kod för klassificering med bästa funktionella approximation

```
pcs_to_include = c(5, 10, 20, 30, 60, 90, 120, 150, 180)

accuracy <- array(dim = length(pcs_to_include))
iter <- 1
for(included_pc in pcs_to_include){
  tail_distance <- array(dim = c(60000,10))
  tail_classification <- array(dim = c(60000))
  for(i in 1:60000){
    for(class_index in 1:10){
      tail_distance[i,class_index] <- sum(principal_coeffs_array[i,class_index,
    included_pc:(dim(principal_coeffs_array)[3])]^2)
    }
    tail_classification[i] <- which.min(tail_distance[i,])
  }
```

```
  accuracy[iter] <- 100*mean((tail_classification-1) == train_labels)
  print(sprintf("%d included PCs: %0.2f percent", included_pc, accuracy[iter]))
  iter <- iter + 1
}

### Testing the PC with best training accuracy on the test data.
best_pc <- pcs_to_include[which.max(accuracy)]

test_distance <- array(dim = c(length(test_labels), 10))
test_classification <- array(dim = c(length(test_labels)))
for(i in 1:length(test_labels)){
  for(class_index in 1:10){
    test_distance[i, class_index] <- sum(test_principal_coeffs_array[i,class_index
    ,best_pc:(dim(test_principal_coeffs_array)[3])]^2)
  }
  test_classification[i] <- which.min(test_distance[i,])
}
test_accuracy <- 100*mean((test_classification-1) == test_labels)
print(sprintf("%d included PCs: %0.2f percent", best_pc, test_accuracy))
```

E.7 Kod för klassificering med bästa vektorapproximation

```
#Note that classwise_pca need to be run with number_of_components_to_save changed
    to 784.
pcs_to_include = c(5, 10, 20, 30, 60, 90, 120, 150, 180, 250, 350, 500)

accuracy <- array(dim = length(pcs_to_include))
iter <- 1
for(included_pc in pcs_to_include){
  tail_distance <- array(dim = c(60000,10))
  tail_classification <- array(dim = c(60000))
  for(i in 1:60000){
    for(class_index in 1:10){
      tail_distance[i,class_index] <- sum(pca_coeffs_array[i,class_index,
    included_pc:(dim(pca_coeffs_array)[3])]^2)
    }
    tail_classification[i] <- which.min(tail_distance[i,])
  }
  accuracy[iter] <- 100*mean((tail_classification-1) == train_labels)
  print(sprintf("%d included PCs: %0.2f percent", included_pc, accuracy[iter]))
  iter <- iter + 1
}

### Testing the PC with best training accuracy on the test data.
best_pc <- pcs_to_include[which.max(accuracy)]

test_distance <- array(dim = c(length(test_labels), 10))
test_classification <- array(dim = c(length(test_labels)))
for(i in 1:length(test_labels)){
  for(class_index in 1:10){
    test_distance[i, class_index] <- sum(test_pca_coeffs_array[i,class_index,
    best_pc:(dim(test_pca_coeffs_array)[3])]^2)
  }
  test_classification[i] <- which.min(test_distance[i,])
}
test_accuracy <- 100*mean((test_classification-1) == test_labels)
print(sprintf("%d included PCs: %0.2f percent", best_pc, test_accuracy))
```

E.8 Kod från Colab, klassificering med neuronnät samt F/-PCA kombinerat med neuronnät

Innehåll

1 Inspektion av datan

[ ]:
```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
print(device_name)
```

```
/device:GPU:0
```

[ ]:
```
!nvidia-smi --query-gpu=gpu_name,driver_version,memory.total --format=csv
```

```
name, driver_version, memory.total [MiB]
Tesla K80, 460.32.03, 11441 MiB
```

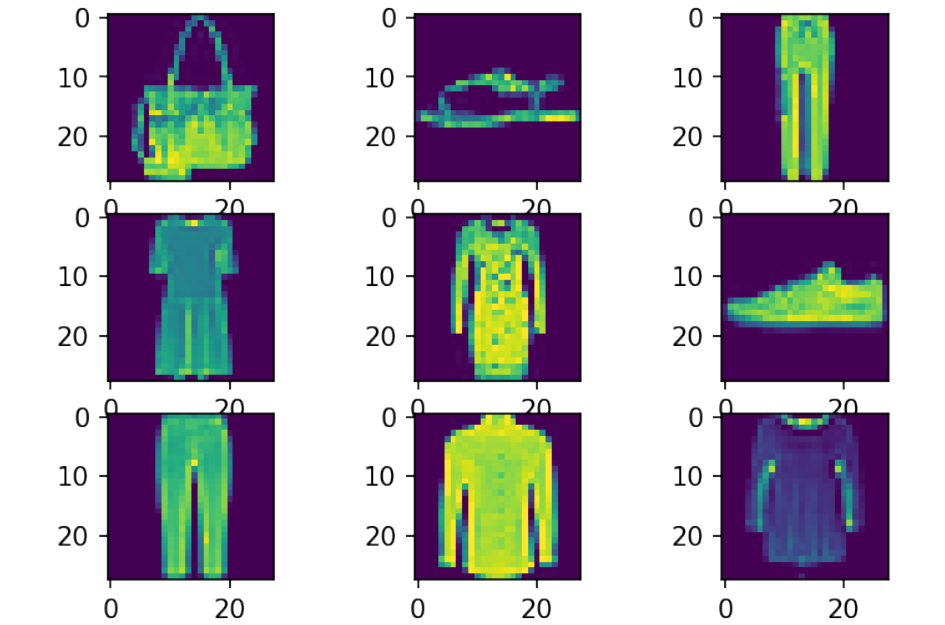
[ ]:
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.
↪load_data()
```

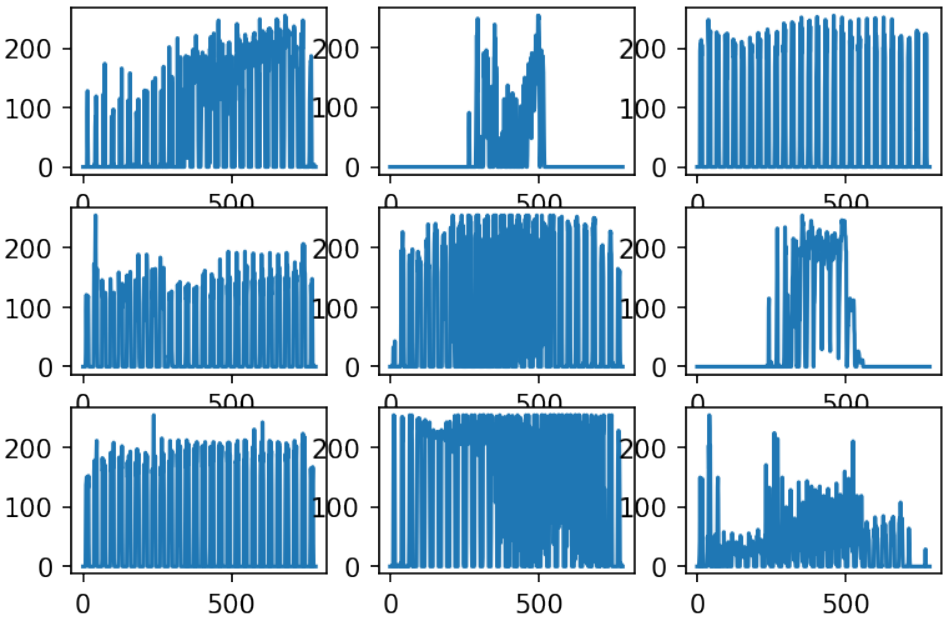
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

[ ]:
```
from matplotlib import pyplot as plt
import numpy as np

print(x_train.shape)
samples_to_show = 9
grid_shape = [3,samples_to_show//3]
sample_index = np.random.choice(60000, samples_to_show)


fig, axes = plt.subplots(grid_shape[0], grid_shape[1], dpi= 150)

axes = axes.reshape(-1)
for i in range(len(axes)):
  axes[i].imshow(x_train[sample_index[i],:])


number_of_pixels = x_train[0,:].size #Should be 28*28
```

```
fig2, axes2 = plt.subplots(grid_shape[0], grid_shape[1], dpi = 150)
axes2 = axes2.reshape(-1)
for i in range(len(axes)):
  axes2[i].plot(np.arange(number_of_pixels), tf.reshape(x_train[sample_index[i],:
 ↪], [number_of_pixels]))
```

```
(60000, 28, 28)
```





2 Klassificering med PCA och FPCA av hela datasetet.

Läser in datan och separerar det i träningsdata och valideringsdata.

[ ]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras

train_coeff_pca = np.load("train_coeff_pca_cut.npy")
eigen_coeffs_matrix_entire_dataset = np.load("eigen_coeffs_matrix_entire_dataset.
 ↪npy")
train_labels = np.load("train_labels.npy")
```

[ ]:
```
train_size = 50000
number_of_eigenvals = 20
```

[ ]:
```
#Only eigencoeffs
train_X_PCA = train_coeff_pca[0:train_size, 0:(number_of_eigenvals)]
val_X_PCA = train_coeff_pca[train_size:60000, 0:(number_of_eigenvals)]

train_X_FPCA = eigen_coeffs_matrix_entire_dataset[0:train_size,0:
 ↪(number_of_eigenvals)]
val_X_FPCA = eigen_coeffs_matrix_entire_dataset[train_size:60000,0:
 ↪(number_of_eigenvals)]
```

Skapar modeller för PCA och FPCA:

[ ]:
```
model_PCA = keras.Sequential([
                        keras.layers.Dense(10*number_of_eigenvals, activation=␣
 ↪'relu', kernel_regularizer = 'l2'),
                        keras.layers.Dense(10*number_of_eigenvals, activation=␣
 ↪'relu', kernel_regularizer = 'l2'),
                        keras.layers.Dense(10, activation = 'softmax')
])
model_PCA.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model_FPCA = keras.Sequential([
                        keras.layers.Dense(10*number_of_eigenvals, activation=␣
 ↪'relu', kernel_regularizer = 'l2'),
                        keras.layers.Dense(10*number_of_eigenvals, activation=␣
 ↪'relu', kernel_regularizer = 'l2'),
                        keras.layers.Dense(10, activation = 'softmax')
])
model_FPCA.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

Tränar modellen på PCA-datan:

[ ]:
```
history_PCA = model_PCA.fit(train_X_PCA, train_labels[0:train_size],␣
↪validation_data = (val_X_PCA, train_labels[train_size:60000]), epochs = 20)
```

Tränar modellen på FPCA-datan:

[ ]:
```
history_FPCA = model_FPCA.fit(train_X_FPCA, train_labels[0:train_size],␣
↪validation_data = (val_X_FPCA, train_labels[train_size:60000]), epochs = 20)
```

Experimentella resultat:
20 epochs, (om man ökar epochs till 40 brukar resultaten bli ~1% bättre)
activation: relu
Layers: Dense 10*#princ. comps.
Dense 10*#princ. comps.
Dense 10, act: softmax
Training accuracy and validation accuracy:
5 pc:
PCA: 65.95%, 65.61%
FPCA: 56.58%, 56.39%
10 pc:
PCA: 75.89%, 74.77%
FPCA: 72.88%, 72.26%
20 pc:
PCA: 82.67%, 81.18%
FPCA: 78.27%, 78.14%?
30 pc:
PCA: 83.80%, 83.56%
FPCA: 81.49%, 80.72%

Layers: Dense 20*#princ. comps.
Dense 20*#princ. comps.
Dense 10, act: softmax
Training accuracy and validation accuracy:
20 pc:
PCA: 82.27%, 81.30%
FPCA: 77.87%, 77.54%

Layers: Dense 10*#princ. comps.
Dense 10*#princ. comps.
Dense 10*#princ. comps.
Dense 10, act: softmax
Training accuracy and validation accuracy:
20 pc:
PCA: 82.35%, 81.68%
FPCA: 77.41%, 76.71%

[ ]:
```
history_PCA.history['val_accuracy'][-1]
```

[ ]:
```
0.8224999904632568
```

3 Korsvalidering av ordinär F-/PCA och neuronnät

[ ]:
```
from sklearn.model_selection import KFold, StratifiedKFold
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras import backend
import gc
train_coeff_pca = np.load("normalized_pca_coeffs_matrix_entire_dataset.npy")
#train_coeff_fpca = np.load("normalized_principal_coeffs_matrix_entire_dataset.
 ↪npy")
train_labels = np.load("train_labels.npy")
#normalized_eigen_coeffs = np.load("normalized_eigen_coeffs_matrix_cut.npy")
```

[ ]:
```
print(train_coeff_pca.shape)
#print(train_coeff_fpca.shape)
#print(eigen_coeffs_matrix_entire_dataset.shape)
#print(normalized_eigen_coeffs.shape)
```

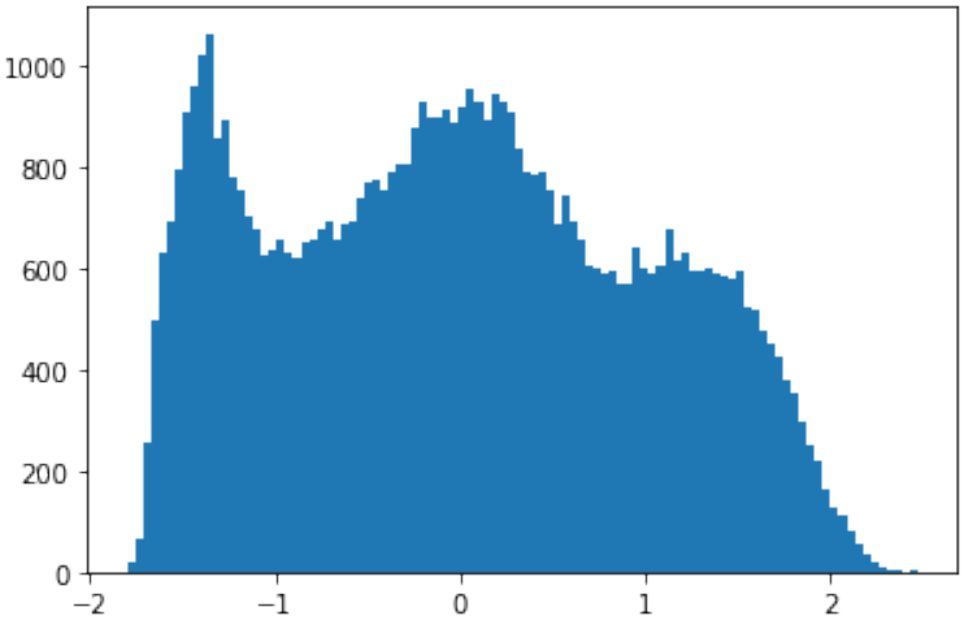
```
(60000, 100)
```

3.0.1 Korsvalidering av PCA och NN

Inspekterar koefficienterna för att se att de har medelvärde 0 och varians 1

[ ]:
```
import matplotlib.pyplot as plt
plt.hist(train_coeff_pca[:,0],bins = 100)
print(np.mean(train_coeff_pca[:,0])) #Should be ~0 if data is correct
print(np.var(train_coeff_pca[:,0], ddof = 1)) #Should be ~1 if data is correct
plt.show()
```

```
1.1202890467150912e-16
1.000000000000002
```



[ ]:
```
K = 10
principal_choices = [5, 10, 20, 30, 40, 60, 80]
#StratifiedKFold bevarar klassbalans, dvs varje klass är ~1/10 av varje fold.
skf = StratifiedKFold(n_splits = K, shuffle = True, random_state=500)

history_PCA = {}




for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  for train_index, test_index in skf.split(train_coeff_pca, train_labels):
    #print("Train: ", train_index, "TEST: ", test_index)
    x_train, x_test = train_coeff_pca[train_index, 0:(number_of_eigenvals)],␣
 ↪train_coeff_pca[test_index, 0:(number_of_eigenvals)]
    y_train, y_test = train_labels[train_index], train_labels[test_index]
    model = None
    model = keras.Sequential(model_layers)
```

```
    model.
↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
    h.append(model.fit(x_train, y_train, validation_data = (x_test,y_test),␣
↪epochs = 10, verbose = 0))
    print("Accuracy with {} PC: {:.2f}%".format(number_of_eigenvals,100*h[-1].
↪history['accuracy'][-1]))
    del model
    backend.clear_session()
    gc.collect()
 history_PCA[number_of_eigenvals] = h
```

```
Accuracy with 5 PC: 72.47%
Accuracy with 5 PC: 73.37%
Accuracy with 5 PC: 73.67%
Accuracy with 5 PC: 73.80%
Accuracy with 5 PC: 74.16%
Accuracy with 5 PC: 74.14%
Accuracy with 5 PC: 74.23%
Accuracy with 5 PC: 74.22%
Accuracy with 5 PC: 74.18%
Accuracy with 5 PC: 74.40%
Accuracy with 10 PC: 79.69%
Accuracy with 10 PC: 80.39%
Accuracy with 10 PC: 80.71%
Accuracy with 10 PC: 80.88%
Accuracy with 10 PC: 81.15%
Accuracy with 10 PC: 81.44%
Accuracy with 10 PC: 81.21%
Accuracy with 10 PC: 81.40%
Accuracy with 10 PC: 81.47%
Accuracy with 10 PC: 81.45%
Accuracy with 20 PC: 83.59%
Accuracy with 20 PC: 84.01%
Accuracy with 20 PC: 84.43%
Accuracy with 20 PC: 84.59%
Accuracy with 20 PC: 84.80%
Accuracy with 20 PC: 84.99%
Accuracy with 20 PC: 84.65%
Accuracy with 20 PC: 85.01%
Accuracy with 20 PC: 84.93%
Accuracy with 20 PC: 85.11%
Accuracy with 30 PC: 84.72%
Accuracy with 30 PC: 85.05%
Accuracy with 30 PC: 85.43%
Accuracy with 30 PC: 85.46%
Accuracy with 30 PC: 85.73%
Accuracy with 30 PC: 85.87%
```

```
Accuracy with 30 PC: 85.77%
Accuracy with 30 PC: 85.90%
Accuracy with 30 PC: 86.08%
Accuracy with 30 PC: 85.97%
Accuracy with 40 PC: 85.30%
Accuracy with 40 PC: 85.86%
Accuracy with 40 PC: 86.02%
Accuracy with 40 PC: 86.09%
Accuracy with 40 PC: 86.48%
Accuracy with 40 PC: 86.53%
Accuracy with 40 PC: 86.33%
Accuracy with 40 PC: 86.64%
Accuracy with 40 PC: 86.66%
Accuracy with 40 PC: 86.59%
Accuracy with 60 PC: 85.99%
Accuracy with 60 PC: 86.35%
Accuracy with 60 PC: 86.68%
Accuracy with 60 PC: 86.69%
Accuracy with 60 PC: 86.77%
Accuracy with 60 PC: 86.85%
Accuracy with 60 PC: 87.04%
Accuracy with 60 PC: 87.20%
Accuracy with 60 PC: 87.11%
Accuracy with 60 PC: 87.04%
Accuracy with 80 PC: 86.45%
Accuracy with 80 PC: 86.74%
Accuracy with 80 PC: 86.99%
Accuracy with 80 PC: 87.03%
Accuracy with 80 PC: 87.32%
Accuracy with 80 PC: 87.32%
Accuracy with 80 PC: 87.10%
Accuracy with 80 PC: 87.36%
Accuracy with 80 PC: 87.49%
Accuracy with 80 PC: 87.45%
```

3.0.2 Korsvalidering av FPCA och NN

[ ]:
```
K = 10
principal_choices = [5, 10, 20, 30, 40, 60, 80]
#StratifiedKFold bevarar klassbalans, dvs varje klass är ~1/10 av varje fold.
skf = StratifiedKFold(n_splits = K, shuffle = True, random_state=500)

history_FPCA = {}
```

```
for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  for train_index, test_index in skf.split(train_coeff_fpca, train_labels):
    #print("Train: ", train_index, "TEST: ", test_index)
    x_train, x_test = train_coeff_fpca[train_index, 0:(number_of_eigenvals)],␣
 ↪train_coeff_fpca[test_index, 0:(number_of_eigenvals)]
    y_train, y_test = train_labels[train_index], train_labels[test_index]
    model = None
    model = keras.Sequential(model_layers)
    model.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
    h.append(model.fit(x_train, y_train, validation_data = (x_test,y_test),␣
 ↪epochs = 10, verbose = 0))
    print("Accuracy with {} PC: {:.2f}%".format(number_of_eigenvals,100*h[-1].
 ↪history['accuracy'][-1]))
    del model
    backend.clear_session()
    gc.collect()
  history_FPCA[number_of_eigenvals] = h
```

```
Accuracy with 5 PC: 72.53%
Accuracy with 5 PC: 73.24%
Accuracy with 5 PC: 73.70%
Accuracy with 5 PC: 73.84%
Accuracy with 5 PC: 73.85%
Accuracy with 5 PC: 73.96%
Accuracy with 5 PC: 74.00%
Accuracy with 5 PC: 74.21%
Accuracy with 5 PC: 74.06%
Accuracy with 5 PC: 74.23%
Accuracy with 10 PC: 79.84%
Accuracy with 10 PC: 80.52%
Accuracy with 10 PC: 80.90%
Accuracy with 10 PC: 81.13%
Accuracy with 10 PC: 81.37%
Accuracy with 10 PC: 81.36%
Accuracy with 10 PC: 81.45%
Accuracy with 10 PC: 81.55%
Accuracy with 10 PC: 81.70%
Accuracy with 10 PC: 81.57%
Accuracy with 20 PC: 83.52%
```

```
Accuracy with 20 PC: 83.92%
Accuracy with 20 PC: 84.26%
Accuracy with 20 PC: 84.37%
Accuracy with 20 PC: 84.81%
Accuracy with 20 PC: 84.96%
Accuracy with 20 PC: 84.91%
Accuracy with 20 PC: 84.95%
Accuracy with 20 PC: 84.98%
Accuracy with 20 PC: 84.99%
Accuracy with 30 PC: 84.64%
Accuracy with 30 PC: 84.94%
Accuracy with 30 PC: 85.14%
Accuracy with 30 PC: 85.15%
Accuracy with 30 PC: 85.47%
Accuracy with 30 PC: 85.61%
Accuracy with 30 PC: 85.57%
Accuracy with 30 PC: 85.67%
Accuracy with 30 PC: 85.93%
Accuracy with 30 PC: 85.80%
Accuracy with 40 PC: 85.24%
Accuracy with 40 PC: 85.41%
Accuracy with 40 PC: 85.83%
Accuracy with 40 PC: 86.03%
Accuracy with 40 PC: 86.29%
Accuracy with 40 PC: 86.31%
Accuracy with 40 PC: 86.41%
Accuracy with 40 PC: 86.42%
Accuracy with 40 PC: 86.45%
Accuracy with 40 PC: 86.58%
Accuracy with 60 PC: 85.80%
Accuracy with 60 PC: 86.15%
Accuracy with 60 PC: 86.62%
Accuracy with 60 PC: 86.54%
Accuracy with 60 PC: 86.74%
Accuracy with 60 PC: 86.87%
Accuracy with 60 PC: 86.82%
Accuracy with 60 PC: 86.97%
Accuracy with 60 PC: 87.09%
Accuracy with 60 PC: 87.14%
Accuracy with 80 PC: 85.71%
Accuracy with 80 PC: 86.24%
Accuracy with 80 PC: 86.56%
Accuracy with 80 PC: 86.48%
Accuracy with 80 PC: 86.79%
Accuracy with 80 PC: 86.74%
Accuracy with 80 PC: 86.92%
Accuracy with 80 PC: 87.11%
Accuracy with 80 PC: 87.14%
```

```
Accuracy with 80 PC: 87.04%
```

3.0.3 Genomsnittliga resultat med korsvalidering av ordinär F-/PCA och neuronnät

[ ]:
```
for pc, h in history_PCA.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
  print("PCA mean CV-accuracy with {} PC:     {:.2f}%,   val_accuracy:   {:.
 ↪2f}%".format(pc, 100*mean_accuracy,100*mean_val_accuracy))
```

```
PCA mean CV-accuracy with 5 PC:     73.86%,   val_accuracy:   73.91%
PCA mean CV-accuracy with 10 PC:     80.98%,   val_accuracy:   80.68%
PCA mean CV-accuracy with 20 PC:     84.61%,   val_accuracy:   84.60%
PCA mean CV-accuracy with 30 PC:     85.60%,   val_accuracy:   85.52%
PCA mean CV-accuracy with 40 PC:     86.25%,   val_accuracy:   86.19%
PCA mean CV-accuracy with 60 PC:     86.77%,   val_accuracy:   86.51%
PCA mean CV-accuracy with 80 PC:     87.13%,   val_accuracy:   86.75%
```

[ ]:
```
for pc, h in history_FPCA.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
  print("FPCA mean CV-accuracy with {} PC:     {:.2f}%,   val_accuracy:   {:.
 ↪2f}%".format(pc, 100*mean_accuracy,100*mean_val_accuracy))
```

```
FPCA mean CV-accuracy with 5 PC:     73.76%,   val_accuracy:   73.87%
FPCA mean CV-accuracy with 10 PC:     81.14%,   val_accuracy:   80.65%
FPCA mean CV-accuracy with 20 PC:     84.57%,   val_accuracy:   84.36%
FPCA mean CV-accuracy with 30 PC:     85.39%,   val_accuracy:   85.31%
FPCA mean CV-accuracy with 40 PC:     86.10%,   val_accuracy:   85.78%
FPCA mean CV-accuracy with 60 PC:     86.67%,   val_accuracy:   86.31%
FPCA mean CV-accuracy with 80 PC:     86.67%,   val_accuracy:   86.37%
```

4 Korsvalidering av klassvis F-/PCA och neuronnät

[ ]:
```
from sklearn.model_selection import KFold, StratifiedKFold
import numpy as np
import tensorflow as tf
from tensorflow import keras

train_labels = np.load("train_labels.npy")
```

```
train_coeff_pca_cw = np.load("normalized_pca_coeffs_matrix.npy")
train_coeff_fpca_cw = np.load("normalized_principal_coeffs_matrix.npy")
print(train_coeff_pca_cw.shape)
print(train_coeff_fpca_cw.shape)
```

```
(60000, 800)
(60000, 800)
```

4.0.1 Korsvalidering av klassvis FPCA och NN:

[ ]:
```
K = 10
principal_choices = [3, 5, 10, 20, 30, 40, 60, 80]
#StratifiedKFold bevarar klassbalans, dvs varje klass är ~1/10 av varje fold.
skf = StratifiedKFold(n_splits = K, shuffle = True, random_state=500)

history_FPCA_Classwise = {}


for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  for train_index, test_index in skf.split(train_coeff_fpca_cw, train_labels):
    #print("Train: ", train_index, "TEST: ", test_index)
    #The 10 first columns corresponds to the largest principal component of each
    x_train, x_test = train_coeff_fpca_cw[train_index, 0:
 ↪(10*number_of_eigenvals)], train_coeff_fpca_cw[test_index, 0:
 ↪(10*number_of_eigenvals)]
    y_train, y_test = train_labels[train_index], train_labels[test_index]
    model = None
    model = keras.Sequential(model_layers)
    model.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
    h.append(model.fit(x_train, y_train, validation_data = (x_test,y_test),␣
 ↪epochs = 10, verbose = 0))
    print("Accuracy with {} PC: {:.2f}%".format(number_of_eigenvals,100*h[-1].
 ↪history['accuracy'][-1]))
    del model
    backend.clear_session()
    gc.collect()
  history_FPCA_Classwise[number_of_eigenvals] = h
for pc, h in history_FPCA_Classwise.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
```

```
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
  print("FPCA mean CV-accuracy with {} PC:     {:.2f}%,   val_accuracy:   {:.
↪2f}%".format(pc, 100*mean_accuracy,100*mean_val_accuracy))
```

```
Accuracy with 3 PC: 78.40%
Accuracy with 3 PC: 80.03%
Accuracy with 3 PC: 80.62%
Accuracy with 3 PC: 80.93%
Accuracy with 3 PC: 81.43%
Accuracy with 3 PC: 81.63%
Accuracy with 3 PC: 81.70%
Accuracy with 3 PC: 81.86%
Accuracy with 3 PC: 81.86%
Accuracy with 3 PC: 81.83%
Accuracy with 5 PC: 81.24%
Accuracy with 5 PC: 82.14%
Accuracy with 5 PC: 82.54%
Accuracy with 5 PC: 82.64%
Accuracy with 5 PC: 83.18%
Accuracy with 5 PC: 83.08%
Accuracy with 5 PC: 83.19%
Accuracy with 5 PC: 83.25%
Accuracy with 5 PC: 83.42%
Accuracy with 5 PC: 83.50%
Accuracy with 10 PC: 83.25%
Accuracy with 10 PC: 83.82%
Accuracy with 10 PC: 84.24%
Accuracy with 10 PC: 84.33%
Accuracy with 10 PC: 84.48%
Accuracy with 10 PC: 84.57%
Accuracy with 10 PC: 84.69%
Accuracy with 10 PC: 84.69%
Accuracy with 10 PC: 84.81%
Accuracy with 10 PC: 84.45%
Accuracy with 20 PC: 84.34%
Accuracy with 20 PC: 84.86%
Accuracy with 20 PC: 85.18%
Accuracy with 20 PC: 85.31%
Accuracy with 20 PC: 85.32%
Accuracy with 20 PC: 85.35%
Accuracy with 20 PC: 85.50%
Accuracy with 20 PC: 85.24%
Accuracy with 20 PC: 85.48%
Accuracy with 20 PC: 85.41%
Accuracy with 30 PC: 84.94%
```

```
Accuracy with 30 PC: 85.35%
Accuracy with 30 PC: 85.51%
Accuracy with 30 PC: 85.55%
Accuracy with 30 PC: 85.85%
Accuracy with 30 PC: 85.75%
Accuracy with 30 PC: 85.76%
Accuracy with 30 PC: 85.87%
Accuracy with 30 PC: 85.78%
Accuracy with 30 PC: 85.74%
Accuracy with 40 PC: 84.94%
Accuracy with 40 PC: 85.16%
Accuracy with 40 PC: 85.70%
Accuracy with 40 PC: 85.55%
Accuracy with 40 PC: 85.79%
Accuracy with 40 PC: 85.73%
Accuracy with 40 PC: 85.69%
Accuracy with 40 PC: 85.68%
Accuracy with 40 PC: 85.70%
Accuracy with 40 PC: 85.66%
Accuracy with 60 PC: 84.98%
Accuracy with 60 PC: 85.18%
Accuracy with 60 PC: 85.33%
Accuracy with 60 PC: 85.44%
Accuracy with 60 PC: 85.71%
Accuracy with 60 PC: 85.63%
Accuracy with 60 PC: 85.78%
Accuracy with 60 PC: 85.70%
Accuracy with 60 PC: 85.68%
Accuracy with 60 PC: 85.67%
Accuracy with 80 PC: 84.78%
Accuracy with 80 PC: 85.29%
Accuracy with 80 PC: 85.31%
Accuracy with 80 PC: 85.38%
Accuracy with 80 PC: 85.48%
Accuracy with 80 PC: 85.69%
Accuracy with 80 PC: 85.40%
Accuracy with 80 PC: 85.54%
Accuracy with 80 PC: 85.64%
Accuracy with 80 PC: 85.58%
FPCA mean CV-accuracy with 3 PC:     81.03%,   val_accuracy:   80.55%
FPCA mean CV-accuracy with 5 PC:     82.82%,   val_accuracy:   82.63%
FPCA mean CV-accuracy with 10 PC:     84.33%,   val_accuracy:   84.22%
FPCA mean CV-accuracy with 20 PC:     85.20%,   val_accuracy:   84.82%
FPCA mean CV-accuracy with 30 PC:     85.61%,   val_accuracy:   85.06%
FPCA mean CV-accuracy with 40 PC:     85.56%,   val_accuracy:   85.45%
FPCA mean CV-accuracy with 60 PC:     85.51%,   val_accuracy:   85.38%
FPCA mean CV-accuracy with 80 PC:     85.41%,   val_accuracy:   85.02%
```

4.0.2 Korsvalidering av klassvis PCA och NN:

[ ]:
```
K = 10
principal_choices = [3, 5, 10, 20, 30, 40, 60, 80]
#StratifiedKFold bevarar klassbalans, dvs varje klass är ~1/10 av varje fold.
skf = StratifiedKFold(n_splits = K, shuffle = True, random_state=500)

history_PCA_Classwise = {}


for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  for train_index, test_index in skf.split(train_coeff_pca_cw, train_labels):
    #print("Train: ", train_index, "TEST: ", test_index)
    #The 10 first columns corresponds to the largest principal component of each
    x_train, x_test = train_coeff_pca_cw[train_index, 0:
 ↪(10*number_of_eigenvals)], train_coeff_pca_cw[test_index, 0:
 ↪(10*number_of_eigenvals)]
    y_train, y_test = train_labels[train_index], train_labels[test_index]
    model = None
    model = keras.Sequential(model_layers)
    model.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
    h.append(model.fit(x_train, y_train, validation_data = (x_test,y_test),␣
 ↪epochs = 10, verbose = 0))
    print("Accuracy with {} PC: {:.2f}%".format(number_of_eigenvals,100*h[-1].
 ↪history['accuracy'][-1]))
    del model
    backend.clear_session()
    gc.collect()
  history_PCA_Classwise[number_of_eigenvals] = h

for pc, h in history_PCA_Classwise.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
  print("PCA mean CV-accuracy with {} PC:     {:.2f}%,   val_accuracy:   {:.
 ↪2f}%".format(pc, 100*mean_accuracy,100*mean_val_accuracy))
```

```
Accuracy with 3 PC: 77.86%
```

```
Accuracy with 3 PC: 79.67%
Accuracy with 3 PC: 80.56%
Accuracy with 3 PC: 80.82%
Accuracy with 3 PC: 81.30%
Accuracy with 3 PC: 81.29%
Accuracy with 3 PC: 81.26%
Accuracy with 3 PC: 81.51%
Accuracy with 3 PC: 81.62%
Accuracy with 3 PC: 81.62%
Accuracy with 5 PC: 81.57%
Accuracy with 5 PC: 82.65%
Accuracy with 5 PC: 83.00%
Accuracy with 5 PC: 82.97%
Accuracy with 5 PC: 83.23%
Accuracy with 5 PC: 83.45%
Accuracy with 5 PC: 83.39%
Accuracy with 5 PC: 83.45%
Accuracy with 5 PC: 83.44%
Accuracy with 5 PC: 83.67%
Accuracy with 10 PC: 83.44%
Accuracy with 10 PC: 83.81%
Accuracy with 10 PC: 84.25%
Accuracy with 10 PC: 84.16%
Accuracy with 10 PC: 84.29%
Accuracy with 10 PC: 84.35%
Accuracy with 10 PC: 84.47%
Accuracy with 10 PC: 84.53%
Accuracy with 10 PC: 84.92%
Accuracy with 10 PC: 84.73%
Accuracy with 20 PC: 84.69%
Accuracy with 20 PC: 85.04%
Accuracy with 20 PC: 85.36%
Accuracy with 20 PC: 85.03%
Accuracy with 20 PC: 85.53%
Accuracy with 20 PC: 85.48%
Accuracy with 20 PC: 85.50%
Accuracy with 20 PC: 85.47%
Accuracy with 20 PC: 85.54%
Accuracy with 20 PC: 85.74%
Accuracy with 30 PC: 85.06%
Accuracy with 30 PC: 85.41%
Accuracy with 30 PC: 85.48%
Accuracy with 30 PC: 85.58%
Accuracy with 30 PC: 85.85%
Accuracy with 30 PC: 85.69%
Accuracy with 30 PC: 85.68%
Accuracy with 30 PC: 85.72%
Accuracy with 30 PC: 85.88%
```

```
Accuracy with 30 PC: 85.73%
Accuracy with 40 PC: 85.17%
Accuracy with 40 PC: 85.58%
Accuracy with 40 PC: 85.79%
Accuracy with 40 PC: 85.73%
Accuracy with 40 PC: 85.88%
Accuracy with 40 PC: 86.20%
Accuracy with 40 PC: 86.01%
Accuracy with 40 PC: 86.11%
Accuracy with 40 PC: 86.10%
Accuracy with 40 PC: 86.09%
Accuracy with 60 PC: 85.53%
Accuracy with 60 PC: 85.67%
Accuracy with 60 PC: 86.01%
Accuracy with 60 PC: 85.82%
Accuracy with 60 PC: 86.16%
Accuracy with 60 PC: 86.03%
Accuracy with 60 PC: 86.03%
Accuracy with 60 PC: 86.14%
Accuracy with 60 PC: 86.14%
Accuracy with 60 PC: 86.11%
Accuracy with 80 PC: 85.49%
Accuracy with 80 PC: 85.78%
Accuracy with 80 PC: 85.76%
Accuracy with 80 PC: 85.72%
Accuracy with 80 PC: 86.17%
Accuracy with 80 PC: 85.81%
Accuracy with 80 PC: 85.81%
Accuracy with 80 PC: 85.91%
Accuracy with 80 PC: 86.08%
Accuracy with 80 PC: 86.01%
PCA mean CV-accuracy with 3 PC:     80.75%,   val_accuracy:   80.75%
PCA mean CV-accuracy with 5 PC:     83.08%,   val_accuracy:   83.02%
PCA mean CV-accuracy with 10 PC:     84.29%,   val_accuracy:   84.22%
PCA mean CV-accuracy with 20 PC:     85.34%,   val_accuracy:   84.95%
PCA mean CV-accuracy with 30 PC:     85.61%,   val_accuracy:   85.36%
PCA mean CV-accuracy with 40 PC:     85.87%,   val_accuracy:   85.73%
PCA mean CV-accuracy with 60 PC:     85.96%,   val_accuracy:   85.64%
PCA mean CV-accuracy with 80 PC:     85.85%,   val_accuracy:   85.73%
```

[ ]:
```
for pc, h in history_PCA_Classwise.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
```

```
  print("PCA mean CV-accuracy with {} PC:     {:.2f}%".format(pc,␣
 ↪100*mean_accuracy))
  print("PCA mean CV-val_accuracy with {} PC: {:.2f}%".format(pc,␣
 ↪100*mean_val_accuracy))
for pc, h in history_FPCA_Classwise.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
  print("FPCA mean CV-accuracy with {} PC:     {:.2f}%".format(pc,␣
 ↪100*mean_accuracy))
  print("FPCA mean CV-val_accuracy with {} PC: {:.2f}%".format(pc,␣
 ↪100*mean_val_accuracy))
```

5 Träning på hela datasetet och med slutgiltig testdata

[ ]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras import backend
import gc
train_coeff_pca = np.load("train_coeff_pca_cut.npy")
train_coeff_fpca = np.load("eigen_coeffs_matrix_entire_dataset.npy")
train_coeff_pca_classwise = np.load("train_coeff_pca_classwise.npy")
train_normalized_eigen_coeffs = np.load("normalized_eigen_coeffs_matrix.npy")

test_coeff_pca = np.load("test_coeff_pca_cut.npy")
test_coeff_fpca = np.load("test_eigen_coeffs_matrix_entire_dataset.npy")
test_coeff_pca_classwise = np.load("test_coeff_pca_classwise.npy")
test_normalized_eigen_coeffs = np.load("normalized_test_eigen_coeffs.npy")
```

[ ]:
```
train_labels = np.load("train_labels.npy")
test_labels = np.load("test_labels.npy")
```

[ ]:
```
print(train_coeff_pca.shape)
print(train_coeff_fpca.shape)
print(train_coeff_pca_classwise.shape)
print(train_normalized_eigen_coeffs.shape)
```

```
(60000, 80)
(60000, 80)
(60000, 800)
(60000, 800)
```

5.0.1 Ordinär PCA och NN

[ ]:
```
train_coeff_pca = np.load("normalized_pca_coeffs_matrix_entire_dataset.npy")
test_coeff_pca = np.load("test_normalized_pca_coeffs_matrix_entire_dataset.npy")
print(np.mean(train_coeff_pca[:,0]))
print(np.var(train_coeff_pca[:,0], ddof = 1))
print(train_coeff_pca.shape)
print(test_coeff_pca.shape)
```

```
1.1202890467150912e-16
1.000000000000002
(60000, 100)
(10000, 100)
```

[ ]:
```
principal_choices = [5, 10, 20, 30, 40, 60, 80]
history_PCA_test = {}

for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  #print("Train: ", train_index, "TEST: ", test_index)
  x_train, x_test = train_coeff_pca[:,0:number_of_eigenvals], test_coeff_pca[:,0:
 ↪number_of_eigenvals]
  y_train, y_test = train_labels, test_labels
  model = None
  model = keras.Sequential(model_layers)
  model.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
  h = model.fit(x_train, y_train, validation_data = (x_test,y_test), epochs =␣
 ↪10, verbose = 0)
  print("Training accuracy with {} PC: {:.2f}%, Test accuracy: {:.2f}%".
 ↪format(number_of_eigenvals,100*h.history['accuracy'][-1], 100*h.
 ↪history['val_accuracy'][-1]))
  history_PCA_test[number_of_eigenvals] = h
  del model
  backend.clear_session()
  gc.collect()
```

```
Training accuracy with 5 PC: 72.61%, Test accuracy: 72.15%
Training accuracy with 10 PC: 79.93%, Test accuracy: 79.32%
Training accuracy with 20 PC: 83.68%, Test accuracy: 82.54%
Training accuracy with 30 PC: 84.73%, Test accuracy: 83.99%
Training accuracy with 40 PC: 85.51%, Test accuracy: 84.47%
```

```
Training accuracy with 60 PC: 85.95%, Test accuracy: 85.09%
Training accuracy with 80 PC: 86.40%, Test accuracy: 84.25%
```

5.0.2 Ordinär FPCA och NN

[ ]:
```
train_coeff_fpca = np.load("normalized_principal_coeffs_matrix_entire_dataset.
 ↪npy")
test_coeff_fpca = np.
 ↪load("test_normalized_principal_coeffs_matrix_entire_dataset.npy")
print(np.mean(train_coeff_fpca[:,0]))
print(np.var(train_coeff_fpca[:,0], ddof = 1))
print(train_coeff_fpca.shape)
print(test_coeff_fpca.shape)
```

```
-4.890902497815356e-17
1.0000000000000009
(60000, 219)
(10000, 219)
```

[ ]:
```
principal_choices = [5, 10, 20, 30, 40, 60, 80]
history_FPCA_test = {}

for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  #print("Train: ", train_index, "TEST: ", test_index)
  x_train, x_test = train_coeff_fpca[:,0:number_of_eigenvals], test_coeff_fpca[:
 ↪,0:number_of_eigenvals]
  y_train, y_test = train_labels, test_labels
  model = None
  model = keras.Sequential(model_layers)
  model.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
  h = model.fit(x_train, y_train, validation_data = (x_test,y_test), epochs =␣
 ↪10, verbose = 0)
  print("Training accuracy with {} PC: {:.2f}%, Test accuracy: {:.2f}%".
 ↪format(number_of_eigenvals,100*h.history['accuracy'][-1], 100*h.
 ↪history['val_accuracy'][-1]))
  history_FPCA_test[number_of_eigenvals] = h
```

```
Training accuracy with 5 PC: 72.68%, Test accuracy: 70.81%
Training accuracy with 10 PC: 79.88%, Test accuracy: 79.30%
```

```
Training accuracy with 20 PC: 83.45%, Test accuracy: 82.37%
Training accuracy with 30 PC: 84.47%, Test accuracy: 83.07%
Training accuracy with 40 PC: 85.10%, Test accuracy: 84.76%
Training accuracy with 60 PC: 85.64%, Test accuracy: 85.07%
Training accuracy with 80 PC: 85.79%, Test accuracy: 85.30%
```

5.0.3 Klassvis PCA och NN

[ ]:
```
train_coeff_pca_classwise = np.load("normalized_pca_coeffs_matrix.npy")
test_coeff_pca_classwise = np.load("test_normalized_pca_coeffs_matrix.npy")
print(np.mean(train_coeff_pca_classwise[:,0]))
print(np.var(train_coeff_pca_classwise[:,0], ddof = 1))
print(train_coeff_pca_classwise.shape)
print(test_coeff_pca_classwise.shape)
```

```
0.6963894040227928
1.0448452619868052
(60000, 800)
(10000, 800)
```

[ ]:
```
principal_choices = [3, 5, 10, 20, 30, 40, 60, 80]
history_PCA_classwise_test = {}

for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  #print("Train: ", train_index, "TEST: ", test_index)
  x_train, x_test = train_coeff_pca_classwise[:,0:(10*number_of_eigenvals)],␣
 ↪test_coeff_pca_classwise[:,0:(10*number_of_eigenvals)]
  y_train, y_test = train_labels, test_labels
  model = None
  model = keras.Sequential(model_layers)
  model.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
  h = model.fit(x_train, y_train, validation_data = (x_test,y_test), epochs =␣
 ↪10, verbose = 0)
  print("Training accuracy with {} PC: {:.2f}%, Test accuracy: {:.2f}%".
 ↪format(number_of_eigenvals,100*h.history['accuracy'][-1], 100*h.
 ↪history['val_accuracy'][-1]))
  history_PCA_classwise_test[number_of_eigenvals] = h
```

```
Training accuracy with 3 PC: 78.13%, Test accuracy: 76.10%
Training accuracy with 5 PC: 81.83%, Test accuracy: 81.06%
```

```
Training accuracy with 10 PC: 83.26%, Test accuracy: 82.43%
Training accuracy with 20 PC: 84.51%, Test accuracy: 83.05%
Training accuracy with 30 PC: 85.10%, Test accuracy: 84.18%
Training accuracy with 40 PC: 85.24%, Test accuracy: 84.53%
Training accuracy with 60 PC: 85.55%, Test accuracy: 84.84%
Training accuracy with 80 PC: 85.49%, Test accuracy: 84.00%
```

5.0.4 Klassvis FPCA och NN

[ ]:
```
train_coeff_fpca_classwise = np.load("normalized_principal_coeffs_matrix.npy")
test_coeff_fpca_classwise = np.load("test_normalized_principal_coeffs_matrix.
 ↪npy")
print(np.mean(train_coeff_fpca_classwise[np.equal(train_labels,0),0]))
print(np.var(train_coeff_fpca_classwise[np.equal(train_labels,0),0], ddof = 1))
print(train_coeff_fpca_classwise.shape)
print(test_coeff_fpca_classwise.shape)
```

```
2.842170943040401e-17
1.0000000000000009
(60000, 800)
(10000, 800)
```

[ ]:
```
K = 10
principal_choices = [3, 5, 10, 20, 30, 40, 60, 80]
history_FPCA_classwise_test = {}

for number_of_eigenvals in principal_choices:
  h = []
  model_layers = [keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(20*number_of_eigenvals, activation= 'relu',␣
 ↪kernel_regularizer = 'l2'),
                  keras.layers.Dense(10, activation = 'softmax')]
  #print("Train: ", train_index, "TEST: ", test_index)
  x_train, x_test = train_coeff_fpca_classwise[:,0:(10*number_of_eigenvals)],␣
 ↪test_coeff_fpca_classwise[:,0:(10*number_of_eigenvals)]
  y_train, y_test = train_labels, test_labels
  model = None
  model = keras.Sequential(model_layers)
  model.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
  h = model.fit(x_train, y_train, validation_data = (x_test,y_test), epochs =␣
 ↪10, verbose = 0)
  print("Training accuracy with {} PC: {:.2f}%, Test accuracy: {:.2f}%".
 ↪format(number_of_eigenvals,100*h.history['accuracy'][-1], 100*h.
 ↪history['val_accuracy'][-1]))
  history_FPCA_classwise_test[number_of_eigenvals] = h
```

```
Training accuracy with 3 PC: 78.81%, Test accuracy: 76.94%
Training accuracy with 5 PC: 81.39%, Test accuracy: 79.92%
Training accuracy with 10 PC: 83.44%, Test accuracy: 82.08%
Training accuracy with 20 PC: 84.54%, Test accuracy: 83.44%
Training accuracy with 30 PC: 85.06%, Test accuracy: 84.53%
Training accuracy with 40 PC: 85.05%, Test accuracy: 82.85%
Training accuracy with 60 PC: 84.88%, Test accuracy: 82.77%
Training accuracy with 80 PC: 84.77%, Test accuracy: 83.25%
```

6 Korsvalidering av FNN och CNN

[ ]:
```
import tensorflow as tf
from sklearn.model_selection import KFold, StratifiedKFold
from tensorflow import keras
from keras import backend
import gc

import math
import numpy as np
import matplotlib.pyplot as plt

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
 ↪load_data()

print("Length of training data is", len(train_images))
print("Length of test data is", len(test_images))
print("Labels sets is of the form", train_labels[1:10])
print("Images is of the following form:", train_images.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
Length of training data is 60000
Length of test data is 10000
Labels sets is of the form [0 0 3 0 2 7 2 5 5]
Images is of the following form: (60000, 28, 28)
```

[ ]:
```
model_layers = {}
model_layers['FNN_modell_1'] = [keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(128, activation='relu'),
                keras.layers.Dense(10, activation='softmax')
                ]
model_layers['FNN_modell_2'] = [keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(88, activation='relu'),
                keras.layers.Dense(40, activation='relu'),
                keras.layers.Dense(10, activation='softmax')
                ]
model_layers['FNN_modell_3'] =  [keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(44, activation='relu'),
                keras.layers.Dense(44, activation='relu'),
                keras.layers.Dense(40, activation='relu'),
                keras.layers.Dense(10, activation='softmax')]
model_layers['FNN_modell_4'] = [keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(20, activation='relu'),
                keras.layers.Dense(20, activation='relu'),
                keras.layers.Dense(10, activation='softmax')]
model_layers['FNN_modell_5'] = [keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(60, activation='relu'),
                keras.layers.Dense(28, activation='relu'),
                keras.layers.Dense(40, activation='relu'),
                keras.layers.Dense(10, activation='softmax')]
model_layers['FNN_modell_6'] = [keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(70, activation='relu'),
                keras.layers.Dense(9, activation='relu'),
                keras.layers.Dense(9, activation='relu'),
                keras.layers.Dense(40, activation='relu'),
                keras.layers.Dense(10, activation='softmax')]


model_layers['CNN_model_1'] = [
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
]
model_layers['CNN_model_2'] = [
```

```
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(50, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
]
model_layers['CNN_model_3'] = [
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(300, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
]
model_layers['CNN_model_4'] = [
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(450, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
]
model_layers['CNN_model_5'] = [
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
]

model_layers['CNN_model_6'] = [
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
```

```
    tf.keras.layers.Dense(64, activation=tf.nn.relu),
    tf.keras.layers.Dense(64, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
]
```

[ ]:
```
K = 10
history_FNN = {}
skf = StratifiedKFold(n_splits = K, shuffle = True, random_state = 500)


for model_name, model_layer in model_layers.items():
  h = []
  for train_index, test_index in skf.split(train_images, train_labels):
    x_train, x_test = train_images[train_index,:,:], train_images[test_index,:,:]
    y_train, y_test = train_labels[train_index], train_labels[test_index]
    if "CNN" in model_name:
      x_train = x_train.reshape(x_train.shape[0],28,28,1)
      x_test = x_test.reshape(x_test.shape[0],28,28,1)
    model = None
    model = keras.Sequential(model_layers[model_name])
    model.compile(optimizer='adam', loss = 'sparse_categorical_crossentropy',␣
 ↪metrics = ['accuracy'])
    h.append(model.fit(x_train,y_train, validation_data = (x_test,y_test),␣
 ↪epochs = 10, verbose = 0))
    print("Accuracy with {}: {:.2f}%, validation accuracy: {:.2f}%".
 ↪format(model_name, 100*h[-1].history['accuracy'][-1],100*h[-1].
 ↪history['val_accuracy'][-1]))
    del model
    backend.clear_session()
    gc.collect()
  history_FNN[model_name] = h

for model_name, h in history_FNN.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
  print("{} CV-accuracy: {:.2f}%, validation accuracy: {:.2f}%".
 ↪format(model_name, 100*mean_accuracy,100*mean_val_accuracy))
```

```
Accuracy with FNN_modell_1: 83.34%, validation accuracy: 82.13%
Accuracy with FNN_modell_1: 84.70%, validation accuracy: 80.80%
Accuracy with FNN_modell_1: 84.58%, validation accuracy: 83.57%
Accuracy with FNN_modell_1: 85.29%, validation accuracy: 84.85%
Accuracy with FNN_modell_1: 85.16%, validation accuracy: 84.90%
Accuracy with FNN_modell_1: 85.81%, validation accuracy: 84.35%
Accuracy with FNN_modell_1: 85.24%, validation accuracy: 85.37%
```

```
Accuracy with FNN_modell_1: 84.60%, validation accuracy: 85.28%
Accuracy with FNN_modell_1: 85.89%, validation accuracy: 84.42%
Accuracy with FNN_modell_1: 85.36%, validation accuracy: 86.02%
Accuracy with FNN_modell_2: 86.11%, validation accuracy: 85.73%
Accuracy with FNN_modell_2: 87.70%, validation accuracy: 87.53%
Accuracy with FNN_modell_2: 88.58%, validation accuracy: 88.15%
Accuracy with FNN_modell_2: 89.03%, validation accuracy: 87.82%
Accuracy with FNN_modell_2: 89.04%, validation accuracy: 88.28%
Accuracy with FNN_modell_2: 89.81%, validation accuracy: 89.02%
Accuracy with FNN_modell_2: 89.34%, validation accuracy: 88.72%
Accuracy with FNN_modell_2: 89.82%, validation accuracy: 89.95%
Accuracy with FNN_modell_2: 89.88%, validation accuracy: 90.50%
Accuracy with FNN_modell_2: 89.00%, validation accuracy: 88.67%
Accuracy with FNN_modell_3: 86.70%, validation accuracy: 86.07%
Accuracy with FNN_modell_3: 88.06%, validation accuracy: 87.42%
Accuracy with FNN_modell_3: 88.77%, validation accuracy: 87.53%
Accuracy with FNN_modell_3: 89.07%, validation accuracy: 88.93%
Accuracy with FNN_modell_3: 89.14%, validation accuracy: 87.88%
Accuracy with FNN_modell_3: 89.16%, validation accuracy: 89.55%
Accuracy with FNN_modell_3: 89.45%, validation accuracy: 88.98%
Accuracy with FNN_modell_3: 89.71%, validation accuracy: 89.17%
Accuracy with FNN_modell_3: 89.96%, validation accuracy: 89.42%
Accuracy with FNN_modell_3: 90.05%, validation accuracy: 89.70%
Accuracy with FNN_modell_4: 86.54%, validation accuracy: 85.23%
Accuracy with FNN_modell_4: 87.98%, validation accuracy: 86.27%
Accuracy with FNN_modell_4: 88.46%, validation accuracy: 88.43%
Accuracy with FNN_modell_4: 89.26%, validation accuracy: 88.87%
Accuracy with FNN_modell_4: 89.48%, validation accuracy: 87.35%
Accuracy with FNN_modell_4: 89.58%, validation accuracy: 88.95%
Accuracy with FNN_modell_4: 90.03%, validation accuracy: 90.05%
Accuracy with FNN_modell_4: 90.25%, validation accuracy: 90.50%
Accuracy with FNN_modell_4: 90.48%, validation accuracy: 89.87%
Accuracy with FNN_modell_4: 90.63%, validation accuracy: 90.48%
Accuracy with FNN_modell_5: 85.48%, validation accuracy: 84.63%
Accuracy with FNN_modell_5: 87.38%, validation accuracy: 87.47%
Accuracy with FNN_modell_5: 88.27%, validation accuracy: 87.52%
Accuracy with FNN_modell_5: 88.90%, validation accuracy: 88.60%
Accuracy with FNN_modell_5: 89.51%, validation accuracy: 88.03%
Accuracy with FNN_modell_5: 89.54%, validation accuracy: 88.97%
Accuracy with FNN_modell_5: 89.79%, validation accuracy: 89.68%
Accuracy with FNN_modell_5: 89.64%, validation accuracy: 86.02%
Accuracy with FNN_modell_5: 90.30%, validation accuracy: 86.62%
Accuracy with FNN_modell_5: 90.27%, validation accuracy: 90.02%
Accuracy with FNN_modell_6: 76.09%, validation accuracy: 74.75%
Accuracy with FNN_modell_6: 84.57%, validation accuracy: 83.85%
Accuracy with FNN_modell_6: 86.21%, validation accuracy: 85.58%
Accuracy with FNN_modell_6: 86.68%, validation accuracy: 85.65%
Accuracy with FNN_modell_6: 87.52%, validation accuracy: 85.85%
```

```
Accuracy with FNN_modell_6: 87.34%, validation accuracy: 86.10%
Accuracy with FNN_modell_6: 88.11%, validation accuracy: 88.23%
Accuracy with FNN_modell_6: 88.45%, validation accuracy: 88.40%
Accuracy with FNN_modell_6: 89.62%, validation accuracy: 89.38%
Accuracy with FNN_modell_6: 90.11%, validation accuracy: 89.72%
Accuracy with CNN_model_1: 95.12%, validation accuracy: 90.75%
Accuracy with CNN_model_1: 97.15%, validation accuracy: 93.27%
Accuracy with CNN_model_1: 98.11%, validation accuracy: 95.42%
Accuracy with CNN_model_1: 98.44%, validation accuracy: 96.50%
Accuracy with CNN_model_1: 98.47%, validation accuracy: 97.07%
Accuracy with CNN_model_1: 98.49%, validation accuracy: 97.42%
Accuracy with CNN_model_1: 98.81%, validation accuracy: 96.70%
Accuracy with CNN_model_1: 98.71%, validation accuracy: 97.93%
Accuracy with CNN_model_1: 98.78%, validation accuracy: 98.27%
Accuracy with CNN_model_1: 98.77%, validation accuracy: 98.37%
Accuracy with CNN_model_2: 94.49%, validation accuracy: 90.93%
Accuracy with CNN_model_2: 96.46%, validation accuracy: 92.67%
Accuracy with CNN_model_2: 97.37%, validation accuracy: 94.32%
Accuracy with CNN_model_2: 97.90%, validation accuracy: 96.43%
Accuracy with CNN_model_2: 98.11%, validation accuracy: 96.10%
Accuracy with CNN_model_2: 98.47%, validation accuracy: 97.03%
Accuracy with CNN_model_2: 98.24%, validation accuracy: 97.40%
Accuracy with CNN_model_2: 98.48%, validation accuracy: 97.23%
Accuracy with CNN_model_2: 98.46%, validation accuracy: 97.88%
Accuracy with CNN_model_2: 98.46%, validation accuracy: 97.98%
Accuracy with CNN_model_3: 95.22%, validation accuracy: 90.23%
Accuracy with CNN_model_3: 97.62%, validation accuracy: 93.37%
Accuracy with CNN_model_3: 98.10%, validation accuracy: 95.45%
Accuracy with CNN_model_3: 98.46%, validation accuracy: 97.07%
Accuracy with CNN_model_3: 98.87%, validation accuracy: 96.28%
Accuracy with CNN_model_3: 98.76%, validation accuracy: 97.77%
Accuracy with CNN_model_3: 98.85%, validation accuracy: 98.28%
Accuracy with CNN_model_3: 99.04%, validation accuracy: 98.00%
Accuracy with CNN_model_3: 99.17%, validation accuracy: 98.32%
Accuracy with CNN_model_3: 99.10%, validation accuracy: 98.85%
Accuracy with CNN_model_4: 95.82%, validation accuracy: 90.25%
Accuracy with CNN_model_4: 97.91%, validation accuracy: 93.88%
Accuracy with CNN_model_4: 98.14%, validation accuracy: 94.38%
Accuracy with CNN_model_4: 98.68%, validation accuracy: 96.95%
Accuracy with CNN_model_4: 98.65%, validation accuracy: 96.45%
Accuracy with CNN_model_4: 98.89%, validation accuracy: 97.90%
Accuracy with CNN_model_4: 98.85%, validation accuracy: 97.55%
Accuracy with CNN_model_4: 99.00%, validation accuracy: 98.37%
Accuracy with CNN_model_4: 99.05%, validation accuracy: 98.43%
Accuracy with CNN_model_4: 99.00%, validation accuracy: 98.38%
Accuracy with CNN_model_5: 95.63%, validation accuracy: 90.18%
Accuracy with CNN_model_5: 97.96%, validation accuracy: 93.90%
Accuracy with CNN_model_5: 98.51%, validation accuracy: 95.53%
```

```
Accuracy with CNN_model_5: 98.65%, validation accuracy: 96.85%
Accuracy with CNN_model_5: 98.79%, validation accuracy: 97.03%
Accuracy with CNN_model_5: 99.05%, validation accuracy: 98.10%
Accuracy with CNN_model_5: 98.89%, validation accuracy: 98.70%
Accuracy with CNN_model_5: 99.13%, validation accuracy: 98.45%
Accuracy with CNN_model_5: 99.12%, validation accuracy: 99.00%
Accuracy with CNN_model_5: 99.34%, validation accuracy: 98.85%
Accuracy with CNN_model_6: 94.27%, validation accuracy: 90.53%
Accuracy with CNN_model_6: 96.83%, validation accuracy: 93.12%
Accuracy with CNN_model_6: 97.83%, validation accuracy: 94.70%
Accuracy with CNN_model_6: 98.22%, validation accuracy: 96.02%
Accuracy with CNN_model_6: 98.04%, validation accuracy: 96.75%
Accuracy with CNN_model_6: 98.34%, validation accuracy: 96.82%
Accuracy with CNN_model_6: 98.58%, validation accuracy: 98.12%
Accuracy with CNN_model_6: 98.64%, validation accuracy: 98.03%
Accuracy with CNN_model_6: 99.28%, validation accuracy: 97.52%
Accuracy with CNN_model_6: 97.94%, validation accuracy: 97.65%
FNN_modell_1 CV-accuracy: 85.00%, validation accuracy: 84.17%
FNN_modell_2 CV-accuracy: 88.83%, validation accuracy: 88.44%
FNN_modell_3 CV-accuracy: 89.01%, validation accuracy: 88.46%
FNN_modell_4 CV-accuracy: 89.27%, validation accuracy: 88.60%
FNN_modell_5 CV-accuracy: 88.91%, validation accuracy: 87.75%
FNN_modell_6 CV-accuracy: 86.47%, validation accuracy: 85.75%
CNN_model_1 CV-accuracy: 98.09%, validation accuracy: 96.17%
CNN_model_2 CV-accuracy: 97.64%, validation accuracy: 95.80%
CNN_model_3 CV-accuracy: 98.32%, validation accuracy: 96.36%
CNN_model_4 CV-accuracy: 98.40%, validation accuracy: 96.25%
CNN_model_5 CV-accuracy: 98.51%, validation accuracy: 96.66%
CNN_model_6 CV-accuracy: 97.80%, validation accuracy: 95.92%
```

[ ]:
```
for model_name, h in history_FNN.items():
  mean_accuracy = 0
  mean_val_accuracy = 0
  for CV_run in h:
    mean_accuracy += CV_run.history['accuracy'][-1]/K
    mean_val_accuracy += CV_run.history['val_accuracy'][-1]/K
  print("{} CV-accuracy: {:.2f}%, validation accuracy: {:.2f}%".
 ↪format(model_name, 100*mean_accuracy,100*mean_val_accuracy))
```

```
FNN_modell_1 CV-accuracy: 85.00%, validation accuracy: 84.17%
FNN_modell_2 CV-accuracy: 88.83%, validation accuracy: 88.44%
FNN_modell_3 CV-accuracy: 89.01%, validation accuracy: 88.46%
FNN_modell_4 CV-accuracy: 89.27%, validation accuracy: 88.60%
FNN_modell_5 CV-accuracy: 88.91%, validation accuracy: 87.75%
FNN_modell_6 CV-accuracy: 86.47%, validation accuracy: 85.75%
CNN_model_1 CV-accuracy: 98.09%, validation accuracy: 96.17%
CNN_model_2 CV-accuracy: 97.64%, validation accuracy: 95.80%
```

```
CNN_model_3 CV-accuracy: 98.32%, validation accuracy: 96.36%
CNN_model_4 CV-accuracy: 98.40%, validation accuracy: 96.25%
CNN_model_5 CV-accuracy: 98.51%, validation accuracy: 96.66%
CNN_model_6 CV-accuracy: 97.80%, validation accuracy: 95.92%
```

7 Bästa modell för FNN och CNN på hela datasetet (ingen korsvalidering)

[ ]:
```
x_train = train_images
x_test = test_images
y_train = train_labels
y_test = test_labels

FNN_modell_4=keras.Sequential([keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(22, activation='relu'),
                keras.layers.Dense(20, activation='relu'),
                keras.layers.Dense(20, activation='relu'),
                keras.layers.Dense(10, activation='softmax')])
FNN_modell_4.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
fit_info = FNN_modell_4.fit(train_images, train_labels, epochs=10,␣
 ↪validation_data = (x_test,y_test))
print("Test accuracy is", FNN_modell_4.evaluate(test_images, test_labels,␣
 ↪verbose=2)[1])
```

```
Epoch 1/10
1875/1875 [==============================] - 9s 4ms/step - loss: 1.1518 -
accuracy: 0.6164 - val_loss: 0.5691 - val_accuracy: 0.8024
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5408 -
accuracy: 0.8098 - val_loss: 0.4929 - val_accuracy: 0.8266
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4775 -
accuracy: 0.8294 - val_loss: 0.4863 - val_accuracy: 0.8258
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4528 -
accuracy: 0.8371 - val_loss: 0.4625 - val_accuracy: 0.8381
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4279 -
accuracy: 0.8453 - val_loss: 0.4481 - val_accuracy: 0.8402
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4106 -
```

```
accuracy: 0.8538 - val_loss: 0.4231 - val_accuracy: 0.8459
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3985 -
accuracy: 0.8579 - val_loss: 0.4805 - val_accuracy: 0.8347
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3943 -
accuracy: 0.8583 - val_loss: 0.4186 - val_accuracy: 0.8514
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3780 -
accuracy: 0.8620 - val_loss: 0.4163 - val_accuracy: 0.8495
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3708 -
accuracy: 0.8670 - val_loss: 0.4482 - val_accuracy: 0.8434
313/313 - 1s - loss: 0.4482 - accuracy: 0.8434
Test accuracy is 0.8434000015258789
```

[ ]:
```
CNN_modell_5=keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), padding='same', activation=tf.nn.relu,
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Conv2D(64, (3,3), padding='same', activation=tf.nn.relu),
    tf.keras.layers.MaxPooling2D((2, 2), strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dense(10,  activation=tf.nn.softmax)
])
CNN_modell_5.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

[ ]:
```
x_train = train_images.reshape(train_images.shape[0],28,28,1)
x_test = test_images.reshape(test_images.shape[0],28,28,1)
y_train = train_labels
y_test = test_labels
```

[ ]:
```
CNN_modell_5.fit(x_train,y_train, epochs = 10, validation_data = (x_test,y_test))
```

```
Epoch 1/10
1875/1875 [==============================] - 40s 7ms/step - loss: 1.7139 -
accuracy: 0.8124 - val_loss: 0.3544 - val_accuracy: 0.8653
Epoch 2/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2802 -
accuracy: 0.8970 - val_loss: 0.2939 - val_accuracy: 0.8953
Epoch 3/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2287 -
accuracy: 0.9130 - val_loss: 0.3145 - val_accuracy: 0.8930
Epoch 4/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1971 -
```

```
accuracy: 0.9254 - val_loss: 0.2948 - val_accuracy: 0.8971
Epoch 5/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1757 -
accuracy: 0.9345 - val_loss: 0.2868 - val_accuracy: 0.9088
Epoch 6/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1538 -
accuracy: 0.9415 - val_loss: 0.3116 - val_accuracy: 0.9027
Epoch 7/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1346 -
accuracy: 0.9504 - val_loss: 0.3527 - val_accuracy: 0.9013
Epoch 8/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1180 -
accuracy: 0.9560 - val_loss: 0.4126 - val_accuracy: 0.8916
Epoch 9/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.1117 -
accuracy: 0.9605 - val_loss: 0.4256 - val_accuracy: 0.9056
Epoch 10/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0947 -
accuracy: 0.9644 - val_loss: 0.4483 - val_accuracy: 0.9059
```

[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x7f23bf8ef310>
```

[ ]:
```
print("Test accuracy is", CNN_modell_5.evaluate(x_test, y_test, verbose=2)[1])
```

```
313/313 - 1s - loss: 0.4483 - accuracy: 0.9059
Test accuracy is 0.9059000015258789
```

8 Inspektion av principalkomponenter

[ ]:
```
import tensorflow as tf
import numpy as np
from tensorflow import keras
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.
 ↪load_data()
```

[ ]:
```
class_index = 9
x_class = x_train[np.equal(y_train,class_index),:,:]
print(x_class.shape)
x_class = x_class.reshape([x_class.shape[0], x_class.shape[1]**2])/255
print(x_class.shape)
class_mean = np.mean(x_class, axis = 0)
print(class_mean.shape)
x_class = x_class - np.mean(x_class, axis = 0)
```

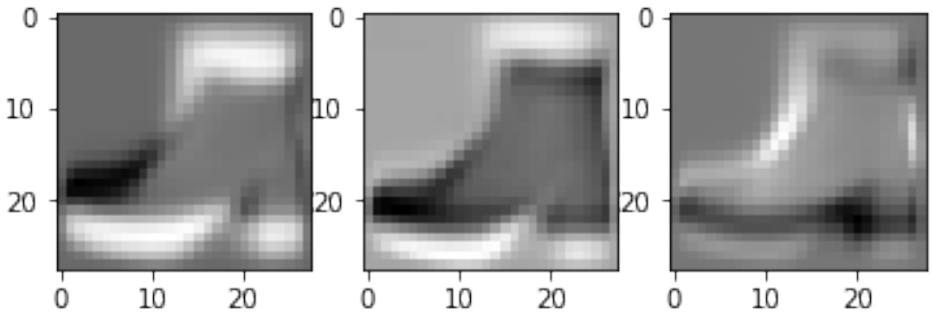
```
(6000, 28, 28)
(6000, 784)
(784,)
```

[ ]:
```
imshow(class_mean.reshape(28,28), cmap = 'gray')
imshow(x_class[590,:].reshape(28,28), cmap = 'gray')
```

[ ]:
```
pca = PCA(svd_solver = 'full')
pca.fit(x_class)
print(pca.components_.shape)
print(pca.explained_variance_[0:10])
```

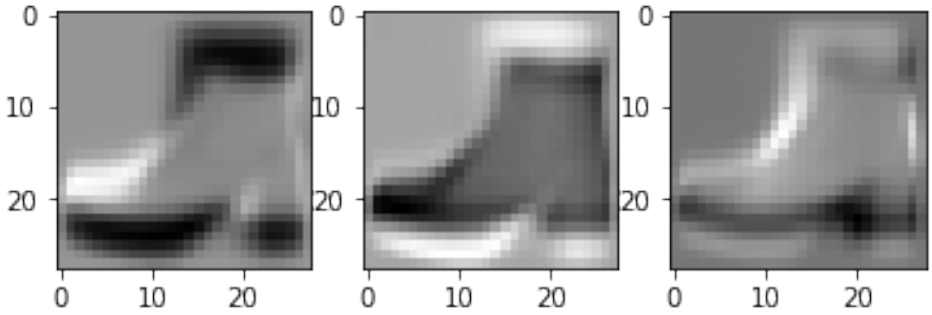
```
(784, 784)
[12.09439907  4.58872545  3.74873471  1.99979476  1.31679144  1.04802688
  0.88884489  0.79067383  0.64199964  0.62015006]
```

[ ]:
```
components_to_view = 3
fig, ax = plt.subplots(nrows = 1, ncols = components_to_view)
for i in range(components_to_view):
  ax[i].imshow(pca.components_[i,:].reshape(28,28), cmap = 'gray')
```



[ ]:
```
x_cov = np.cov(np.transpose(x_class))
print(x_cov.shape)
w, v = np.linalg.eig(x_cov)

print(w[0:10])
components_to_view = 3
fig, ax = plt.subplots(nrows = 1, ncols = components_to_view)
for i in range(components_to_view):
  ax[i].imshow(np.real(v[:,i]).reshape(28,28), cmap = 'gray')
#print(np.real(v[:,0]) + pca.components_[:,0])
```

```
(784, 784)
[12.09439907+0.j  4.58872545+0.j  3.74873471+0.j  1.99979476+0.j
  1.31679144+0.j  1.04802688+0.j  0.88884489+0.j  0.79067383+0.j
  0.64199964+0.j  0.62015006+0.j]
```

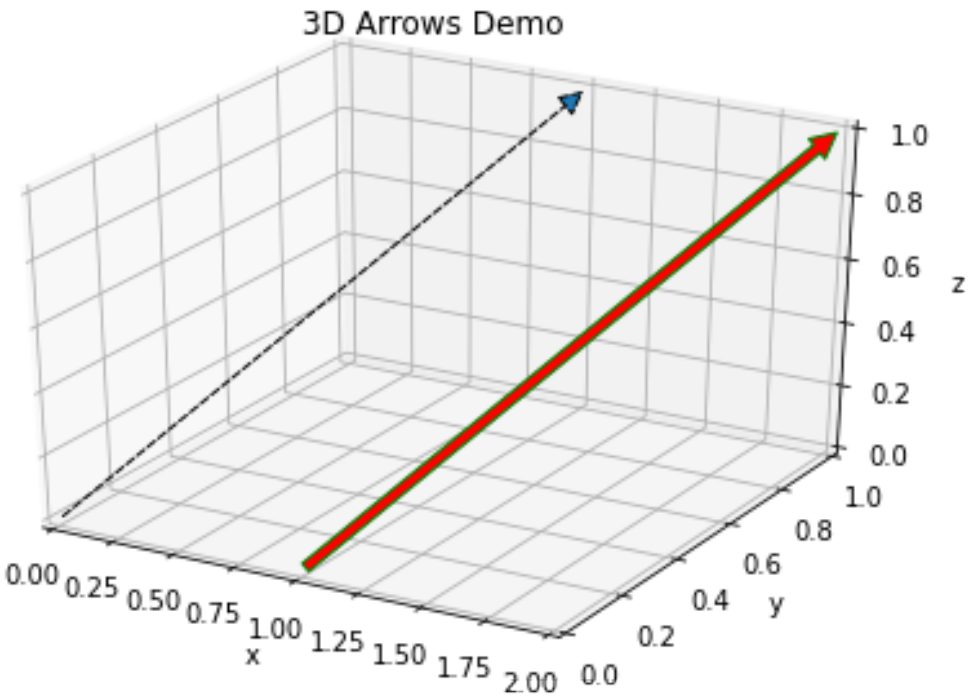


9 Visualisering av PCA

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
#From: https://gist.github.com/WetHat/1d6cd0f7309535311a539b42cccca89c
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.proj3d import proj_transform
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib.patches import FancyArrowPatch
class Arrow3D(FancyArrowPatch):

    def __init__(self, x, y, z, dx, dy, dz, *args, **kwargs):
        super().__init__((0, 0), (0, 0), *args, **kwargs)
        self._xyz = (x, y, z)
        self._dxdydz = (dx, dy, dz)

    def draw(self, renderer):
        x1, y1, z1 = self._xyz
        dx, dy, dz = self._dxdydz
        x2, y2, z2 = (x1 + dx, y1 + dy, z1 + dz)

        xs, ys, zs = proj_transform((x1, x2), (y1, y2), (z1, z2), self.axes.M)
        self.set_positions((xs[0], ys[0]), (xs[1], ys[1]))
        super().draw(renderer)

def _arrow3D(ax, x, y, z, dx, dy, dz, *args, **kwargs):
    '''Add an 3d arrow to an `Axes3D` instance.'''

    arrow = Arrow3D(x, y, z, dx, dy, dz, *args, **kwargs)
    ax.add_artist(arrow)
```

```
setattr(Axes3D, 'arrow3D', _arrow3D)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlim(0,2)
ax.arrow3D(0,0,0,
           1,1,1,
           mutation_scale=20,
           arrowstyle="-|>",
           linestyle='dashed')
ax.arrow3D(1,0,0,
           1,1,1,
           mutation_scale=20,
           ec ='green',
           fc='red')
ax.set_title('3D Arrows Demo')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
fig.tight_layout()
```



[3]:
```
numberOfSamples = 400
samplesInPCA = np.random.normal(0,1,[400,3])
variances = [np.sqrt(10), np.sqrt(3), np.sqrt(0.3)]
samplesInPCA = samplesInPCA * variances
```

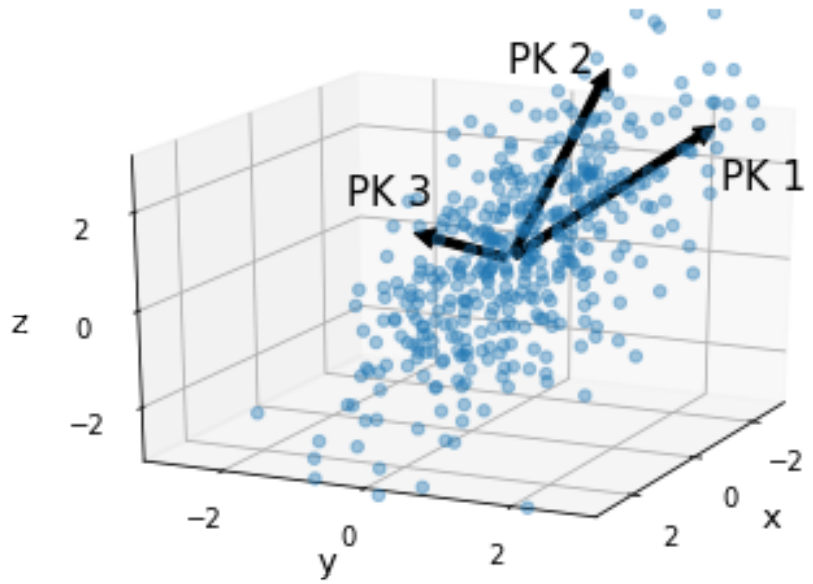
```
PCAvectors = np.zeros([3,3])
PCAvectors[:,0] = [1,1,1]
PCAvectors[:,1] = [-1,0,1]
PCAvectors[:,2] = [1,-2,1]

for i in range(3):
  PCAvectors[:,i] = PCAvectors[:,i]/np.linalg.norm(PCAvectors[:,i])
print(samplesInPCA)
print(PCAvectors)
```

```
[[11.10538602 -1.36922283  0.29064907]
 [ 1.09415543  1.16436695  0.42073884]
 [ 0.66096259  2.61611859  0.70585629]
 ...
 [ 2.76468565  2.40635393  0.37058987]
 [ 1.43916281 -1.41260973  0.13176086]
 [ 2.3120071   1.70785169  0.43554702]]
[[ 0.57735027 -0.70710678  0.40824829]
 [ 0.57735027  0.         -0.81649658]
 [ 0.57735027  0.70710678  0.40824829]]
```

[4]:
```
centerPosition = np.transpose(np.array([1,1,1], ndmin = 2))
samplesPositions = np.matmul(PCAvectors,np.transpose(samplesInPCA)) +␣
 ↪centerPosition

fig = plt.figure()
ax = fig.gca(projection = '3d')

ax.view_init(20,25)
u = PCAvectors[0,:]*variances
v = PCAvectors[1,:]*variances
w = PCAvectors[2,:]*variances
#ax.quiver(np.ones(3), np.ones(3), np.ones(3), u, v, w, arrow_length_ratio=0.
 ↪3,length = 3, color = "orange")
ax.set_xlim3d(-3,3)
ax.set_ylim3d(-3,3)
ax.set_zlim3d(-3,3)
ax.set_xlabel("x", fontsize = 13)
ax.set_ylabel("y", fontsize = 13)
ax.set_zlabel("z", fontsize = 13)
ax.set_xticks([-2,0,2])
ax.set_yticks([-2,0,2])
ax.set_zticks([-2,0,2])

for i in range(3):
  scale_factor = 2.5
```

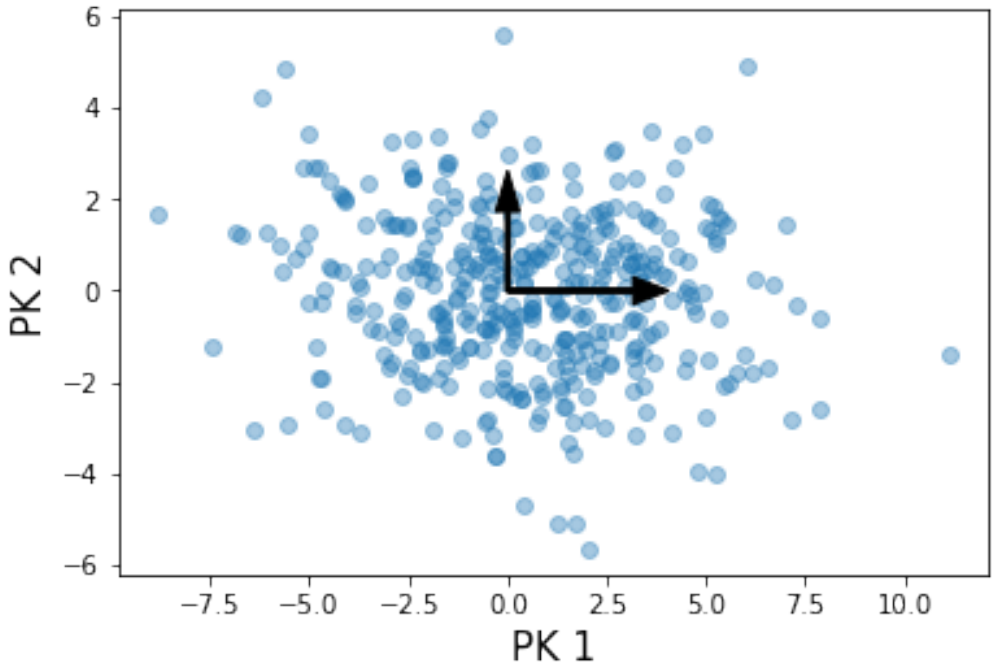
```
  dx = PCAvectors[0,i]*scale_factor*variances[i]
  dy = PCAvectors[1,i]*scale_factor*variances[i]
  dz = PCAvectors[2,i]*scale_factor*variances[i]
  ax.arrow3D(centerPosition[0,0], centerPosition[1,0], centerPosition[2,0],␣
 ↪dx,dy,dz, mutation_scale = 15, color = "black")
ax.scatter(samplesPositions[0,:], samplesPositions[1,:], samplesPositions[2,:],␣
 ↪alpha = 0.4)

textpos1 = centerPosition[:,0] + PCAvectors[:,0]*8
ax.text(textpos1[0], textpos1[1],textpos1[2]-1.2, "PK 1", fontsize = 15, color =␣
 ↪"black")
textpos2 = centerPosition[:,0] + PCAvectors[:,1]*4
ax.text(textpos2[0], textpos2[1]-1.35,textpos2[2], "PK 2", fontsize = 15, color␣
 ↪= "black")
textpos3 = centerPosition[:,0] + PCAvectors[:,2]*3
ax.text(textpos3[0], textpos3[1]+0.8,textpos3[2], "PK 3", fontsize = 15, color =␣
 ↪"black")
#ax.text(3,3,3, "PC 1", PCAvectors[:,0], fontsize = 15, color = "red")
print(samplesPositions.shape)
plt.savefig("PCA_3D.pdf")
```

```
(3, 400)
```



[5]:
```
fig = plt.figure()
ax = fig.gca()

ax.scatter(samplesInPCA[:,0],samplesInPCA[:,1], alpha = .4)
```

```
ax.arrow(0, 0, variances[0], 0, width = 0.1, color = 'black', head_width = 0.6)
ax.arrow(0, 0, 0, variances[1], width = 0.1, color = 'black', head_width = 0.6)
ax.set_xlabel("PK 1", fontsize = 15)
ax.set_ylabel("PK 2", fontsize = 15)
plt.savefig("PCA_2D.pdf")
```



9.1 Two classes

[6]:
```
numberOfSamples = 400
samplesInPCA = np.random.normal(0,1,[2,400,3])
variances = np.array([[np.sqrt(10), np.sqrt(3), np.sqrt(0.3)],[np.sqrt(10), np.
↪sqrt(3), np.sqrt(0.3)]])
samplesInPCA = samplesInPCA * np.expand_dims(variances,axis = 1)

PCAvectors = np.zeros([2,3,3])
PCAvectors[0,:,0] = [1,1,1]
PCAvectors[0,:,1] = [-1,0,1]
PCAvectors[0,:,2] = [1,-2,1]

PCAvectors[1,:,0] = [1,-3,1]
PCAvectors[1,:,1] = [1,1,2]
PCAvectors[1,:,2] = np.cross(PCAvectors[1,:,0],PCAvectors[1,:,1])

for i in range(3):
```

```
  for j in range(2):
    PCAvectors[j,:,i] = PCAvectors[j,:,i]/np.linalg.norm(PCAvectors[j,:,i])
print(samplesInPCA)
print(PCAvectors)
```

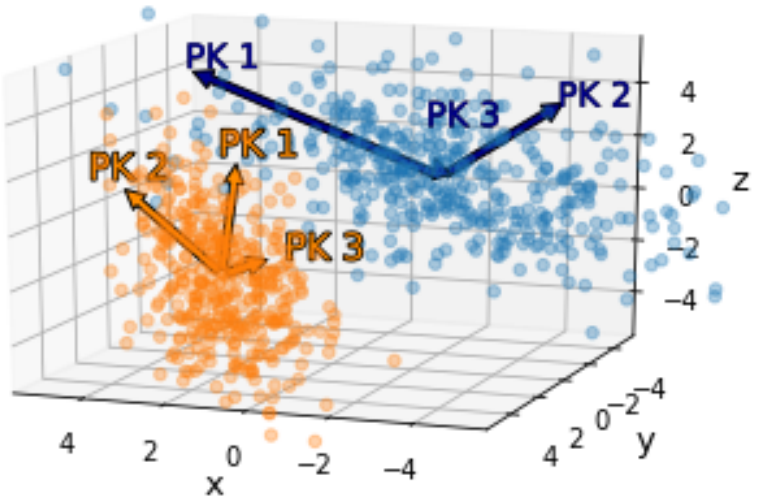
```
[[[ 0.65218113 -0.98405561  0.00646347]
  [-2.04663055 -2.16056718 -0.62164384]
  [-1.93052654 -0.06221467  0.26360732]
  ...
  [ 1.361199    1.57284986  0.60150882]
  [ 5.22876302  0.13094071 -0.56270978]
  [ 0.37071664  0.22696456 -0.11865628]]

 [[-3.73037818  1.92265709 -0.08516529]
  [-3.61658587 -2.81648733  0.22973815]
  [-1.56320242 -1.41441844  0.5210155 ]
  ...
  [ 4.20941158  0.67788234 -0.45460771]
  [ 5.71258704 -0.66621271 -0.57529742]
  [ 4.42465804  1.33578817 -0.17152749]]]
[[[ 0.57735027 -0.70710678  0.40824829]
  [ 0.57735027  0.         -0.81649658]
  [ 0.57735027  0.70710678  0.40824829]]

 [[ 0.30151134  0.40824829 -0.86164044]
  [-0.90453403  0.40824829 -0.12309149]
  [ 0.30151134  0.81649658  0.49236596]]]
```

[7]:
```
import matplotlib.patheffects as PathEffects
centerPosition = np.transpose(np.array([[-2,-2,1],
                                        [2,2,-2]], ndmin = 2))
samplesPositions = np.zeros([2,3,numberOfSamples])
print(samplesPositions.shape)
print(variances.shape)
print(centerPosition.shape)
for j in range(2):
  samplesPositions[j,:,:] = np.matmul(PCAvectors[j,:,:],np.
 ↪transpose(samplesInPCA[j,:,:])) + np.expand_dims(centerPosition[:,j],1)

fig = plt.figure()
ax = fig.gca(projection = '3d')
width = 5.5
ax.set_xlim3d(-width,width)
ax.set_ylim3d(-width,width)
ax.set_zlim3d(-width,width)
ax.set_xlabel("x", fontsize = 13)
ax.set_ylabel("y", fontsize = 13)
```

```
ax.set_zlabel("z", fontsize = 13)
ax.set_xticks([-4,-2,0,2,4])
ax.set_yticks([-4,-2,0,2,4])
ax.set_zticks([-4,-2,0,2,4])
#ax.quiver(np.ones(3), np.ones(3), np.ones(3), u, v, w, arrow_length_ratio=0.
 ↪3,length = 3, color = "orange")

ax.view_init(15,110)
colors = ['darkblue','darkorange']
for j in range(2):
  ax.scatter(samplesPositions[j,0,:], samplesPositions[j,1,:],␣
 ↪samplesPositions[j,2,:], alpha = 0.35)

for j in range(2):
  u = PCAvectors[j,0,:]*variances[j,:]
  v = PCAvectors[j,1,:]*variances[j,:]
  w = PCAvectors[j,2,:]*variances[j,:]
  for i in range(3):
    scale_factor = 2.5
    dx = PCAvectors[j,0,i]*scale_factor*variances[j,i]
    dy = PCAvectors[j,1,i]*scale_factor*variances[j,i]
    dz = PCAvectors[j,2,i]*scale_factor*variances[j,i]
    ax.arrow3D(centerPosition[0,j], centerPosition[1,j], centerPosition[2,j],␣
 ↪dx,dy,dz, mutation_scale = 15, color = colors[j], alpha = 1, ec = 'black')
txt = [None,]*3
textpos1 = centerPosition[:,0] + PCAvectors[0,:,0]*8
txt[0] = ax.text(textpos1[0], textpos1[1],textpos1[2], "PK 1", fontsize = 13,␣
 ↪color = colors[0])
textpos2 = centerPosition[:,0] + PCAvectors[0,:,1]*4
txt[1] = ax.text(textpos2[0], textpos2[1],textpos2[2], "PK 2", fontsize = 13,␣
 ↪color = colors[0])
textpos3 = centerPosition[:,0] + PCAvectors[0,:,2]*3
txt[2] = ax.text(textpos3[0], textpos3[1],textpos3[2], "PK 3", fontsize = 13,␣
 ↪color = colors[0])
#ax.text(3,3,3, "PC 1", PCAvectors[:,0], fontsize = 15, color = "red")
for t in txt:
  t.set_path_effects([PathEffects.withStroke(linewidth=0.5, foreground='black')])
textpos1 = centerPosition[:,1] + PCAvectors[1,:,0]*8
txt[0] = ax.text(textpos1[0]+0.5, textpos1[1],textpos1[2], "PK 1", fontsize =␣
 ↪14, color = colors[1])
textpos2 = centerPosition[:,1] + PCAvectors[1,:,1]*4
txt[1] = ax.text(textpos2[0]+1, textpos2[1],textpos2[2]+0.4, "PK 2", fontsize =␣
 ↪14, color = colors[1])
textpos3 = centerPosition[:,1] + PCAvectors[1,:,2]*1.7
txt[2] = ax.text(textpos3[0], textpos3[1],textpos3[2]-0.2, "PK 3", fontsize =␣
 ↪14, color = colors[1])
```

```
for t in txt:
  t.set_path_effects([PathEffects.withStroke(linewidth=1.5, foreground='black')])

print(samplesPositions.shape)
plt.savefig("PCA_3D_two_classes.pdf",bbox_inches = 'tight')
```

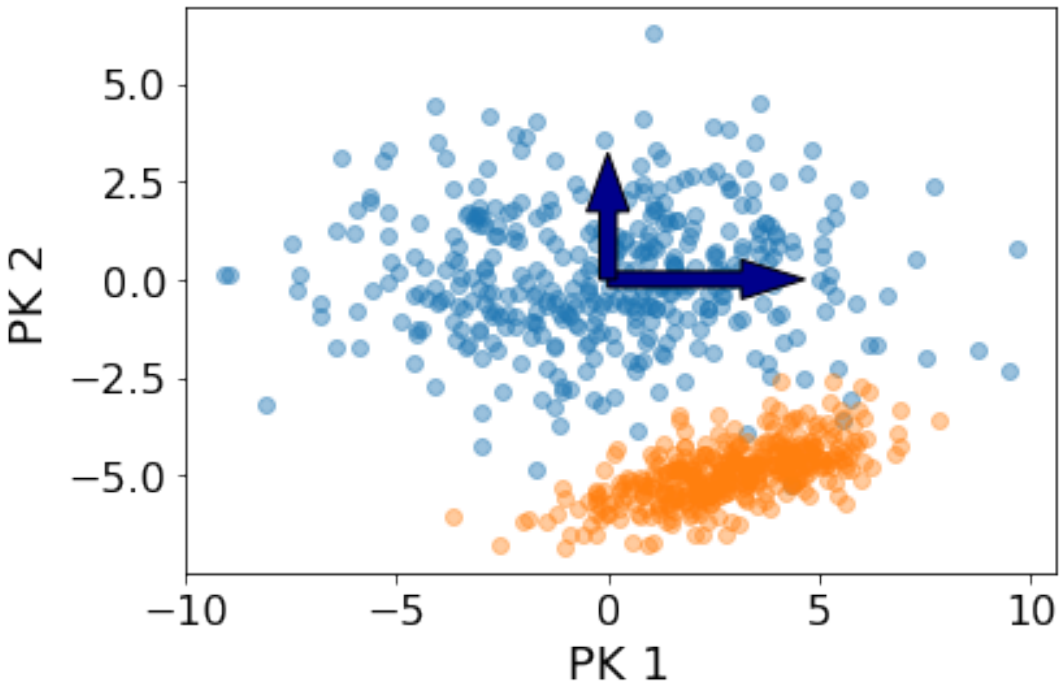
```
(2, 3, 400)
(2, 3)
(3, 2)
(2, 3, 400)
```



[8]:
```
fig = plt.figure()
ax = fig.gca()
print((samplesPositions[1,:,:] - np.expand_dims(centerPosition[:,0],1)).shape)
otherClassSamples = np.matmul(np.transpose(PCAvectors[0,:,:
 ↪]),(samplesPositions[1,:,:] - np.expand_dims(centerPosition[:,0],1)))
print(otherClassSamples.shape)

ax.scatter(samplesInPCA[0,:,0],samplesInPCA[0,:,1], alpha = .45)
ax.scatter(otherClassSamples[0,:],otherClassSamples[1,:], alpha = .4)
ax.arrow(0, 0, variances[0,0], 0, width = 0.4, color = colors[0], head_width =␣
 ↪1, ec = 'black')
ax.arrow(0, 0, 0, variances[0,1], width = 0.4, color = colors[0], head_width =␣
 ↪1, ec = 'black')
ax.set_xlabel("PK 1", fontsize = 18)
ax.set_ylabel("PK 2", fontsize = 18)
ax.tick_params(axis = 'both', labelsize = 16)
```

```
plt.savefig("PCA_2D_two_classes_first.pdf", bbox_inches = 'tight')
```

```
(3, 400)
(3, 400)
```



[9]:
```
fig = plt.figure()
ax = fig.gca()
print((samplesPositions[1,:,:] - np.expand_dims(centerPosition[:,0],1)).shape)
otherClassSamples = np.matmul(np.transpose(PCAvectors[1,:,:
 ↪]),(samplesPositions[0,:,:] - np.expand_dims(centerPosition[:,1],1)))
print(otherClassSamples.shape)

ax.scatter(otherClassSamples[0,:],otherClassSamples[1,:], alpha = .45)
ax.scatter(samplesInPCA[1,:,0],samplesInPCA[1,:,1], alpha = .35)
ax.arrow(0, 0, variances[1,0], 0, width = 0.4, color = colors[1], head_width =␣
 ↪1, ec = 'black')
ax.arrow(0, 0, 0, variances[1,1], width = 0.4, color = colors[1], head_width =␣
 ↪1, ec = 'black')
ax.set_xlabel("PK 1", fontsize = 18)
ax.set_ylabel("PK 2", fontsize = 18)

ax.tick_params(axis = 'both', labelsize = 16)
plt.savefig("PCA_2D_two_classes_second.pdf", bbox_inches = 'tight')
```

```
(3, 400)
(3, 400)
```