1.1 Wat *tensors* zijn en voorbeelden voor de dimensies tm 5D Tensor: vector of (2D) matrix van vectoren, (3D) (batch, hoogte, breedte), 4D (batch, hoogte, breedt, kanalen), 5D (batch, frames, hoogte, breedte, kanalen) bijv. van videostreaming 1.2 hoe neurale netwerken zijn opgebouwd (linear + activation), en hoe ze een extensie zijn van lineaire modellen (wanneer kun je ze stapelen)
de functie y = f[x, params] is een opsommig van een line van de type Wx+b 1.3 neurale netwerken zijn een universal function Een universele functiebenadering (universal function approximation) is een concept in de wiskunde en machine learning dat verwijst naar het vermogen van een model om elke continue functie te benaderen, mits het model voldoende complexiteit heeft. Het idee is dat een model (zoals een neuraal netwerk) in staat is om elke willekeurige functie te benaderen tot een gewenste nauwkeurigheid, gegeven voldoende parameters, lagen, of tijd om te trainen. Universal Approximation Theorem: een feedforward neuraal netwerk met minstens één hidden layer (met voldoende aantal neuronen) kan een continue functie kan benaderen, op elke gesloten en begrensde interval, tot elke gewenste nauwkeurigheid. 1.4 gradient descent: een optimalisatie-algoritme gebruikt in machine learning en deep learning om de parameters van een model (zoals de gewichten in een neuraal netwerk) aan te passen en zo de fout (of verlies) te minimaliseren.

1.5 voordelen van een datastreamer voor grote datasets: data worden in batch aangeboden, om data in real-time of in kleine stukken (data chunks) te verwerken, in plaats van het laden van de volledige dataset in het geheugen. Het biedt voordelen zoals efficiënt geheugengebruik, schaalbaarheid, real-time verwerking, en kostenbesparing, terwijl het ook in staat is om te werken met gedistribueerde gegevens en lage latentie biedt voor directe besluitvorming. **Batch:** mechanisme om meer willekeur te krijgen. Elke iteratie de algoritme krijt een subset van de training data en berekent de gradient op basis van deze subset en niet de gehele dataset. Dit minibatch kan helpen om lokale minima te voorkomen en meer te richten naar global minima, wat meer generalisatie van de output functie betekent. De batch wordt gebruikt in de Stochastic gradient descent (SGD). Als de batch gelijk is aan de gehele dataset dan is de SGD gelijk aan de normale gradient descent. Een hele bewerking van de dataset is een epoch. Voordelen SGD: voegt noise omdat de berekening op een kleine subset wordt gedaan, kan de lokale minima escapen 1.6 Wat de **invloed is van het aantal units in een dense layer** Meer units in een dense layer geven het model meer capaciteit om complexe relaties in de gegevens te leren. Elke unit in een laag kan als een soort "kenmerkdetector" worden gezien, dus meer units maken het model krachtiger in het herkennen van complexere patronen. Als je te weinig units gebruikt, kan het model onderfitten (underfitting), wat betekent dat het model niet in staat is om de onderliggende patronen in de gegevens goed te leren. Als je te veel units gebruikt, kan het model overfitten (overfitting), wat betekent dat het model te veel leert van de specifieke details in de trainingsgegevens en slecht generaliseert naar nieuwe, ongeziene gegevens. Network capacity: aantal hidden units, meer hidden units meer benadering van complexe functies (width), hidden units zijn nodes Network depth: aantal layers

Meer units leiden tot een groter aantal parameters in het netwerk, wat betekent dat er meer rekenkracht en geheugen nodig is voor zowel de training als de inferentie (voorspelling). Dit verhoogt de trainingstijd, omdat het netwerk meer berekeningen moet uitvoeren tijdens elke stap van de gradient descent-optimalisatie. De benodigde tijd voor backpropagation en gewicht-updates neemt toe naarmate het aantal units in een laag toeneemt. Wanneer we modellen trainen, zoeken we naar de parameters die de best mogelijke mapping/koppeling van input naar output voor de betreffende taak produceren. De loss function geeft een mate van mismatch tussen voorspelling en output. We zoeken parameters die dit veschil zo klein mogelijk maken zodat de input gemapt is aan de output met zo min mogelijk verschil. Parameters kunnen zijn: intercept of slope, d.w.z. de slope waarden == gewichten

1.7 **Hoe een gin file werkt** en wat de voordelen daarvan zijn: in een gin file worden de hyperparameters van een neurale netwerk mee gegeven bij het aanmaken van een netwerk. Daar kunnen de kenmerk ook aangepast worden voor hypertuning. 1.8 Waarom we **train-test-valid splits** maken Trainingsset: Wordt gebruikt om het model te trainen. Validatieset: Wordt gebruikt om het model te evalueren en hyperparameters af te stemmen. Testset: Wordt pas gebruikt na het trainen en afstemmen van het model om de uiteindelijke prestaties te testen. Een gebruikelijke verdeling is bijvoorbeeld: 70% voor training 15% voor validatie 15% voor testen Hyperparameters zijn instellingen van het model die niet tijdens het trainen worden geleerd, zoals de leersnelheid, het aantal lagen in een neuraal netwerk, het aantal eenheden in elke laag, de batchgrootte, en meer. Je kunt de validatieset gebruiken om verschillende instellingen van hyperparameters uit te proberen en te testen om te zien welke het beste presteren. Dit proces wordt vaak hyperparameteroptimalisatie genoemd. Een veelgebruikte aanpak om hyperparameters af te stemmen is grid search of random search, waarbij je verschillende combinaties van hyperparameters test en de configuratie kiest die de beste resultaten oplevert op de validatieset. **Validatie**: Nadat het model is getraind, wordt de nauwkeurigheid geëvalueerd op de validatieset. Dit helpt ons te begrijpen hoe goed het model presteert met de gegeven hyperparameters. **Hyperparameteroptimalisatie**: Bijv. we testen verschillende leersnelheden (learning_rate_init) om te zien welke het beste werkt op de validatieset. Het model met de hoogste nauwkeurigheid op de validatieset wordt beschouwd als het beste model. **Testen**: Het beste model wordt uiteindelijk geëvalueerd op de testset. De testset wordt pas gebruikt nadat de hyperparameters zijn geoptimaliseerd en het model is getraind op de trainingsset.

1.9 Wat een Activation function is, en kent er een aantal (ReLU, Leaky ReLU, ELU, GELU, Sigmoid) Een **activation function** (activeringsfunctie) is een wiskundige functie die de output van een neuron in een neuraal netwerk bepaalt. Het zorgt ervoor dat het netwerk niet-lineair (non-linear) wordt, waardoor het in staat is om complexe patronen en relaties in de data te leren. *ReLU* is de meest gebruikte activatiefunctie in diepe netwerken. Het geeft de input door als de waarde groter is dan 0, en anders geeft het 0 terug. Het is eenvoudig en leidt tot snellere training, maar kan het probleem van "dode neuronen" veroorzaken, waarbij neuronen nooit geactiveerd worden als ze negatieve inputs ontvangen. *Leaky ReLU* is een variant van ReLU die voorkomt dat neuronen "doodgaan" door negatieve waarden een kleine, niet-nul output te geven (in plaats van 0). Dit zorgt ervoor dat de neuron ook voor negatieve inputs actief blijft. *ELU* is een activatiefunctie die voor positieve waarden werkt zoals ReLU, maar voor negatieve waarden wordt de output een exponentiële functie die de negatieve output verzacht. Het helpt bij het verminderen van de bias van de activatiefunctie en het versnellen van de convergentie in vergelijking met ReLU. *GELU* is een activatiefunctie die een combinatie is van een sigmoïde en een ReLU, en het heeft een gladder verloop dan ReLU. Het is gebaseerd op een probabilistische benadering en wordt vaak gebruikt in Transformer-netwerken zoals BERT. GELU is wiskundig geavanceerder dan ReLU, maar werkt vaak beter in moderne netwerken. De **Sigmoidfunctie** is een S-vormige (sigmoïde) curve die de output beperkt tot het interval tussen 0 en 1. Het wordt vaak gebruikt in de outputlaag van classificatiemodellen voor binaire classificatie, omdat de output kan worden geïnterpreteerd als de waarschijnlijkheid van een klasse. ReLU: Eenvoudig, snel, maar kan leiden tot "dode" neuronen. Leaky ReLU: Voorkomt dode neuronen door een kleine negatieve helling te geven. ELU: Een exponentiële variant die negatieve inputs verzacht, waardoor sneller convergeren mogelijk is. GELU: Een gladde, probabilistische activatiefunctie die goed werkt in moderne netwerken zoals Transformers. Sigmoid: Beperkt de output tussen 0 en 1, vaak gebruikt voor binaire classificatie. 1.10 wat een loss functie is: Een loss functie (of kostenfunctie) berekent de fout of het verschil meet tussen de voorspelde waarden van een model en de werkelijke waarden (targets) in een dataset Mean Squared Error (MSE): Toepassing: Veel gebruikt voor regressieproblemen. **MSE** berekent het gemiddelde van de kwadraten van de verschillen tussen de werkelijke en voorspelde waarden. Dit betekent dat grotere fouten zwaarder worden gestraft, wat kan helpen om grote afwijkingen te vermijden. **Cross-Entropy Loss**(ook bekend als Log Loss): Deze loss functie meet de afwijking tussen de werkelijke klasse

en de voorspelde waarschijnlijkheid. Toepassing: Veel gebruikt voor classificatieproblemen, vooral binaire en multi-class classificatie. Afhankelijk van het type probleem (bijvoorbeeld classificatie of regressie) kies je een geschikte loss functie zoals Mean Squared Error (MSE) voor regressie of Cross-Entropy Loss voor classificatie.
1.11 Hoe deep learning past binnen de geschiedenis van AI, en wat deep learning kenmerkt ten opzichte van de rest. First neuron idea is from 1943, perceptron (1958), 1980's backpropagatio developed which started a new development
1.12 **stappen van het trainen van een NN*-: datapreparatie, trainbare gewichten, predict, lossfunctie, optimizers A supervised learning model is a fuction y = f[x, params] that relates input x to input y by parameters. To train the model we define a loss function over a training dataset which quantifies the mismatch between the prediction of f and the observed outputs y. We search the parameters that minimize the loss. We evaluate on a different set of data (test) to see how well it generalizes.

LES 2

2.1 - Verschillende loss functies (MSE en RMSE, Negative Log Likelihood, Cross Entropy Loss, Binary Cross Entropy loss) en in welke gevallen je ze moet gebruiken of vermijden:

RMSE

MSE (for regression models):

Mean Squard Error, neemt de som van alle verschillen n tussen y en predicted y tot de macht 2 (alleen positief), en gedeeld door het aantal data n. Het is default voor regressie loss:

$$MSE = \frac{1}{n}\sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

This is the mean $\frac{1}{n}\sum_{i=1}^n$ of the squared error $(Y_i - \hat{Y}_i)^2$ But torch has already implemented that for us in an optimized way:

```
loss = torch.nn.MSELoss()
loss(yhat, y)
```

Negative log likelihood.

Likelihood: In probabilistische modellen beschrijft likelihood hoe waarschijnlijk het is dat een bepaald set van parameters de geobserveerde data heeft gegenereerd. Stel je voor dat je een model hebt dat een kansverdeling voorspelt voor een bepaalde dataset. De likelihood is de kans dat je de werkelijke observaties ziet, gegeven de parameters van je model. De **Negative Log Likelihood** is simpelweg de negatieve waarde van de logaritme van de likelihood. De reden dat we de log gebruiken, is om te zorgen dat de berekeningen eenvoudiger zijn (de kans op meerdere gebeurtenissen wordt vermenigvuldigd, maar de log van een product is de som van de logs, wat rekentechnisch handiger is). Het "negatieve" teken wordt toegevoegd omdat we vaak een verliesfunctie willen minimaliseren tijdens de training van het model. Log-likelihood is echter een maat voor de waarschijnlijkheid (die we willen maximaliseren), dus door het negatief te maken, veranderen we het in een verliesfunctie die we willen minimaliseren. The function is:

$$NLL = - log(\hat{y}[c])$$

Or: take the probabilities $\hat{y}$, and pick the probability of the correct class c from the list of probabilities with $\hat{y}[c]$. Now take the log of that.

The log has the effect that predicting closer to 0 if it should have been 1 is punished extra.

```
loss = torch.nn.NLLLoss()
loss(yhat, y)
```

Het is te bewijzen dat negative log-likehood en cross entropy equivalent zijn. **Cross Entropy Loss**: Cross-entropy meet de "afstand" tussen twee waarschijnlijkheidsdistributies: De echte distributie (meestal de werkelijke labels, vaak één-hot gecodeerd). De voorspelde distributie (de waarschijnlijkheden die het model toekent aan de verschillende klassen). Het meet het verschil tussen twee waarschijnlijkheidsdistributies: de voorspelde waarschijnlijkheden en de werkelijke waarschijnlijkheden (de echte labels). In eenvoudige termen berekent de Cross Entropy Loss hoe goed de voorspelde klassen waarschijnlijkheden overeenkomen met de werkelijke klassenlabels == vind het minimum verschil in waarchijnlijkheid. Als de voorspelde waarschijnlijkheid dicht bij 1 ligt voor de juiste klasse (en dicht bij 0 voor de andere klassen), zal het verlies klein zijn (omdat og(1)=0). Als de voorspelde waarschijnlijkheid ver van 1 ligt (d.w.z. onjuiste voorspellingen), wordt het verlies groter, wat het model aanmoedigt om zich aan te passen in de volgende iteraties. Categorical CEL: output is een verdeling van probabiliteit per categorie. LogSoftmax is dan ook geintegreerd.

Binary Cross Entropy loss: output is een waarde tussen 0 en 1 (wel of niet de categorie) **momenutum**: updates the parameers with a weighted combination of the gradient computed from the current batch en the direction moved in the prevous step. The weights of the past gradient gets smaller. **adam**: normalize the gradients so we move a fixed distance governed by the learning rate in each direction. This to avoid that big gradients gets big changes en small gradients get small changes (because of the calculation without normalization). **L2 regularization**: applied to weights but not biases, weight decay term. The effect is to encourage smaller weights so the output function is smoother and less overfitted. Small weight mean a small variance and therefore the error reduces because the network do not overfit the data. When the netwok is overparametirized the regularization term will favour functions that interpolate smoothly between nerby points. 2.2 - Dimensionality probleem is bij afbeeldingen van Dense Neural Networks: fully connected networks hebben het nadeel dat ze niet goed werken voor grote inputs zoals beelden waarin de neuronen samen specifieke waarnemingen moeten nemen, er is geen nut dat elke neuroon meedoet met elke onderdeel. Eerder is dat handiger om een beeld in parallel te analiseren en steeds in grotere gebieden te integreren. (p. 51). Verder shallow dense networks zijn trager in het trainen en kunnen minder goed generaliseren. 2.3 - Hoe Convolutions dat dimensionality probleem oplossen: beelden hebben een hoge dimensie (bij. 224x224 maken in tot 150.528 input dimensies). Hidden layers in fully connected networks zijn groter dan de input, dit kan betekenen meer dan 22 miljard neurone. Pixels van beelden die dichtbij zijn,zijn ook statistisch aan elkaar verbonden. FC moeten patronen leren per pixel en in elke mogelijke positie terwijl bij beelden spraken is van samengestelde patronen in verschillende posities. CNN leren door gebruik te maken van minder parameters (gewichtjes) en kunnen omgaan met geaugmenteerde beelden (rotaties, uitgestrekt, gespiegeld enz.) en kunnen hierdoor patronen beter abstraheren. **Filter (convolutional kernel)**: convolutional layer met dezelfde gewichten op elke positie, berekent een gewogen som van de dichtbij inputs. Kernel size kan verhoogt worden maar blijf een oneven aantal zodat gecentreerd is op de huidige pixel positie. Grotere kernel== meer gewichten. Via dilatation rate kunnen 0 tussen de gewichten toegevoegd worden zodat er een grotere kernel gebuikt wordt maar niet meer gewichten. stride: (stap), stride = 1, elke positie word door de kernel berekend (groote blijft gelijk), stride =2 berekende de helft van de pixels

convolutional layer: berekent de output door de input te bewerken en de bias toe te voegen en elke resultaat via een activatiefunctie door te geven. Elke convolutie genereert a nieuwe set van hidden variable (feature map of channel). Example: Kernel and Convolution in Action Let’s say we have a 3x3 kernel and a 5x5 input image:

Input Image (5x5):

[1 2 3 0 1 4 5 6 1 2 7 8 9 2 3 1 3 5 1 4 2 4 6 0 1]

Kernel (3x3):

[1 0 −1 1 0 −1 1 0 −1] Now, let's apply the convolution operation with a stride of 1 and no padding. Starting from the top-left corner, we calculate the dot product between the kernel and the input image at that position. The first dot product is:

$(1*1)+(0*2)+(-1*3)+(1*4)+(0*5)+(-1*6)+(1*7)+(0*8)+(-1*9)=-6$ This value (-6) will be placed in the output feature map. The kernel then moves to the next position, sliding over by one pixel (because of stride 1), and the process is repeated until the entire image is processed.

De kernel in een CNN is een kleine matrix die over de invoergegevens beweegt en convolutie-operaties uitvoert om belangrijke kenmerken te extraheren. De kernel is verantwoordelijk voor het detecteren van specifieke patronen in de gegevens, en meerdere kernels worden meestal gebruikt om verschillende kenmerken op verschillende abstractieniveaus in een netwerk te detecteren. Deze geleerde patronen of kenmerken worden vervolgens doorgegeven aan diepere lagen voor verdere verwerking en classificatie.

2.4 - Wat maxpooling doet: vermindert de dimensies door een gemiddelde of een maximum over een pool size (window) te berekenen. Max pooling vermindert de ruimtelijke dimensies van de feature map, terwijl de belangrijkste kenmerken behouden blijven door de maximumwaarde uit een venster van waarden te nemen. Het helpt de grootte van het netwerk te verkleinen, verhoogt de rekenkundige efficiëntie en verbetert de translatie-invariantie. Stride bepaalt hoeveel de pooling-window beweegt bij elke stap. De output feature map na pooling heeft kleinere ruimtelijke dimensies, maar **behoudt de diepte (aantal kanalen)**.

2.5 - Wat een stride en padding zijn
2.6 - Wat een Convolution van 1x1 filter doet: Het 1x1 filter wordt toegepast op elke pixel van de input feature map, om de channels te verandere. Berekend een gewogen som van alle channels op een pixel positie. Imdat de filtermaat 1x1 is, kijkt het alleen naar de waarden van één pixel per keer en hierdoor is het mogelijk om de diepte (aantal kanalen) van de feature map te veranderen. Lineaire transformatie: De 1x1 filter voert een lineaire transformatie uit door een gewichten matrix toe te passen op de verschillende kanalen (diepte) van de input. Hierdoor kunnen nieuwe, gecombineerde representaties van de input worden gemaakt zonder de ruimtelijke dimensies te beïnvloeden. 2.7 - Kent ruwweg de innovaties die de verschillende architecturen doen: AlexNet, VGG, GoogleNet, ResNet, SENet Alexnet: eerste CNN die goed werkte (op Imagenet dataset). Vorzien van 8 hidden layers, Relu, eerste 5 convoluitonals en rest FC. VGG: dieper dan Alexnet, 18 lagen GoogleNet: hier werd de 1x1 kernel bedacht, nog diepere netwerk. Dit toonde wel aan dat meer lagen niet altijd beter performance opleverden. Dit heeft te maken met weight decay, die is groter naar mate meer lagen er zijn. Een kleine weiziging in de beginlagen kan grote gevolgen hebben op diepere lagen. (shattered gradients) ResNet: residual or skip connection zijn aftakkingen van een berekening waar de input van elke layer is weer toegevoegd aan de output op recursieve wijzen. De residual connection generre een ensambel van kleinere netwerken en hun output is samengevoegd (opgeteld) in een resultaat. 2.8 - Begrijpt wat overfitting is en hoe regularisatie (batchnorm, split van training/test/validation, dropout, learning rate) helpt.

De student kan:
2.9 - reproduceren hoe een Convolution schaalt (O(n)) ten opzicht van een NN (O(n^2)) De schaling van een Convolutional Neural Network (CNN) in vergelijking met een Fully Connected Neural Network (FCNN) kan worden begrepen door naar het aantal berekeningen of parameters te kijken die nodig zijn om een input te verwerken.

1. Fully Connected Neural Network (FCNN) Schaling In een fully connected neural network (FCNN), ook wel een dense network, zijn alle neuronen in elke laag verbonden met alle neuronen in de volgende laag. Dit betekent dat het aantal verbindingen (parameters/gewichten) snel toeneemt naarmate je het aantal neuronen vergroot.

Stel dat je een inputvector hebt van lengte n en dat je een verborgen laag hebt met m neuronen. De matrixvermenigvuldiging tussen de inputvector en de gewichtenmatrix van deze laag heeft O(n * m) berekeningen nodig. Als je meerdere lagen hebt, wordt de tijdcomplexiteit van het hele netwerk O(n^2) als het aantal lagen in het netwerk lineair toeneemt.

Voorbeeld:

Stel je voor dat je een netwerk hebt met 1000 inputneuronen en 1000 neuronen in de eerste verborgen laag. De aantal parameters (gewichten) is dan 1000 * 1000 = 1.000.000. Als je een netwerk hebt met 10 lagen, neemt het aantal berekeningen exponentieel toe, omdat elke laag met elke andere laag volledig verbonden is. De tijdcomplexiteit kan O(n^2) worden, afhankelijk van het aantal neuronen in elke laag. 2. Convolutional Neural Network (CNN) Schaling In een Convolutional Neural Network (CNN), daarentegen, worden de berekeningen op een andere manier uitgevoerd. CNN's gebruiken convoluties om lokale patronen in de input te detecteren, wat resulteert in een betere schaling dan in een volledig verbonden netwerk. Dit komt omdat een convolutioneel filter maar een klein gedeelte van de input hoeft te verwerken (in plaats van de volledige input), en dit filter kan herhaaldelijk over de input worden toegepast om kenmerken te extraheren.

De schaling van een CNN wordt meestal beschreven als O(n), omdat de convolutionele filters de input lokaal verwerken en geen volledige matrixvermenigvuldiging nodig hebben zoals in een FCNN.

Waarom is het O(n) voor een CNN? Convolutie (filter toepassen): Stel dat je een 2D afbeelding van grootte n x n hebt (bijvoorbeeld 28x28 pixels), en je past een klein filter toe, bijvoorbeeld een 3x3 filter. In plaats van dat elke pixel met elke andere pixel wordt verbonden (zoals bij een volledig verbonden netwerk), wordt het filter slechts op een lokaal gebied toegepast (bijvoorbeeld 3x3 van de pixels) en schuift het filter door de afbeelding.

Bij elke toepassing van het filter op een 3x3 regio, worden er slechts 9 berekeningen uitgevoerd. Het filter beweegt over de afbeelding met een bepaalde stapgrootte (stride), en daarom is het aantal convoluties (en dus de berekeningen) lineair gerelateerd aan de inputgrootte O(n).
Gewichten (Weights): Dit zijn specifieke parameters die de verbindingen tussen neuronen in een neuraal netwerk vertegenwoordigen. Parameters: Dit is de algemene term die alle waarden omvat die het netwerk leert, inclusief gewichten én biases.

2.10 - Een configureerbaar Convolution NN bouwen en handmatig hypertunen
2.11 - MLFlow gebruiken naast gin-config
2.12 - een Machine Learning probleem classificeren als classificatie / regressie / reinforcement / (semi)unsupervised

2.13 - begrijpt welke dimensionaliteit een convolution nodig heeft en wat de strategieen zijn om dit te laten aansluiten op andere tensors

Adjusting the Number of Filters Gradually increasing filters: It's common practice to start with a small number of filters in the first layer and progressively increase them in subsequent layers. This is because the lower layers typically capture simpler, more general features (e.g., edges, colors), while the higher layers capture more complex, abstract features. In the context of Convolutional Neural Networks (CNNs), filters (also known as kernels) are small matrices or weights that are used to scan (or convolve) over the input data in order to detect specific features. These features can be patterns like edges, textures, or shapes in the input, and they are learned during the training process.

LES 3

3.1 - **motivatie is om RNNs te gebruiken**: RNNs zijn speciaal ontworpen om sequentiële gegevens te verwerken, zoals tijdseries, tekst, spraak of andere gegevens waarbij de volgorde van belang is. Tijdreeksanalyse: In toepassingen zoals weersvoorspelling, aandelenmarktvoorspelling of sensor monitoring, zijn de gegevens afhankelijk van voorgaande tijdstappen. RNNs kunnen de context van eerdere tijdstappen behouden, wat ze geschikt maakt voor deze taken.

1. Geheugen voor Vorige Informatie RNNs hebben een intern geheugen (door middel van hun terugkoppeling) waarmee ze informatie van eerdere tijdstappen kunnen onthouden. Dit is essentieel voor taken zoals:

Taalmodellering: Waarbij de betekenis van een zin of tekst vaak afhangt van de vorige woorden. Spraakherkenning: Waarbij het systeem de context van vorige geluidsgolven moet begrijpen om de betekenis van de huidige geluidsgolf te interpreteren. 3. Flexibiliteit bij Lange en Korte Afhankelijkheden RNNs kunnen zowel korte als lange termijn afhankelijkheden in de gegevens begrijpen. Ze kunnen de relatie tussen gegevens in opeenvolgende tijdstappen leren, wat cruciaal is voor bijvoorbeeld:

Vertalen van zinnen: Waar de vertaling van een zin afhankelijk is van woorden of zinsdelen die ver in de zin liggen. Handschriftherkenning: Waar de context van eerdere tekens nodig is om de huidige te begrijpen.

5. End-to-End Leerbaarheid In tegenstelling tot traditionele benaderingen waarbij vooraf gedefinieerde regels of handmatige kenmerken nodig zijn, kunnen RNNs leren van ruwe gegevens. Dit betekent dat ze de eind-naar-eind taak kunnen leren, zoals: Vertalen van teksten zonder vooraf gedefinieerde grammaticale regels en samenvatten van documenten door automatisch de belangrijkste informatie te extraheren.
6. Gebruik in Diverse Toepassingen RNNs worden op grote schaal gebruikt in verschillende domeinen, zoals: Natural Language Processing (NLP): Voor tekstvertaling, samenvatting, sentimentanalyse, en spraakherkenning. Tijdreeksvoorspelling: Voor toepassingen in de financiën, gezondheidszorg en weerpatronen. Beeldbeschrijving: Voor het genereren van beschrijvingen van beelden (door sequenties van woorden te genereren op basis van beelden).

3.2 - wat een window en horizon zijn: **Window** (Venster): Het aantal voorgaande tijdstappen (gegevenspunten) die het model op elk moment gebruikt voor zijn berekeningen. **Horizon**: Het aantal toekomstige tijdstappen dat het model probeert te voorspellen of te voorspellen. 3.3 - Wat belangrijk is bij datapreparatie om data leakage te voorkomen: leakage komt bij timeseries voor als de voorspellingen van een periode al gebaseerd zijn op data in latere tijdstippen 3.4 - Hoe een simple RNN werkt (hidden state): In tegenstelling tot traditionele feedforward neurale netwerken, kunnen RNNs informatie uit het verleden

behouden en gebruiken voor de verwerking van de huidige input. De informatie wordt in een hidden layer bewaard en meegegeven naar de volgende layers, naar mate van de vordering is wordt de begin informatie vergeten en de meest recenten informatie onthoueden. 3.5 - Waarin een GRU en LSTM verschillen van RNN (gates): GRU (Gated Recurrent Unit) en LSTM (Long Short-Term Memory) zijn beide varianten van Recurrent Neural Networks (RNNs), ontworpen om enkele van de beperkingen van traditionele RNNs te overwinnen, zoals het probleem van vanishing gradients bij lange sequenties. LSTM heeft twee gates die bepalen welke info belangrij is voor onthouden en welke vergeten mag worden. De GRU heeft maar een gate, update gate, die bepaald welke info doorgegeven wordt.

GRU (Gated Recurrent Unit) Simpele structuur: GRU heeft een eenvoudiger ontwerp dan LSTM. Het maakt gebruik van twee gates: **Update gate (reset gate)**: Dit regelt hoeveel van de vorige toestand wordt behouden en hoeveel van de nieuwe informatie wordt toegevoegd. **Reset gate**: Het bepaalt hoeveel van de vorige geheugentoestand moet worden "vergeten". Aantal gates: GRU heeft minder parameters dan LSTM omdat het maar twee gates gebruikt. Dit maakt GRU sneller om te trainen en efficiënter in gebruik van geheugen. **LSTM (Long Short-Term Memory)** Complexe structuur: LSTM heeft drie gates die de stroom van informatie door de tijdstappen regelen: **Forget gate**: Regelt hoeveel van de vorige toestand moet worden vergeten. **Input gate**: Regelt hoeveel nieuwe informatie moet worden toegevoegd aan de toestand. **Output gate**: Regelt welke informatie uit de toestand naar de volgende laag wordt doorgestuurd. Meer complexiteit: LSTM heeft meer parameters omdat het drie gates bevat in plaats van twee. Dit maakt het krachtiger dan GRU in sommige gevallen, maar het kan ook langzamer en geheugenintensiever zijn om te trainen.

Belangrijkste verschillen tussen RNN, GRU en LSTM

Eigenschap	RNN	GRU	LSTM
Aantal Gates	Geen (basis RNN heeft geen gates)	2 gates (update, reset)	3 gates (forget, input, output)
Complexiteit	Eenvoudig, maar slecht in lange afhankelijkheden	Minder complex dan LSTM, maar krachtiger dan RNN	Complexer dan GRU, maar krachtig voor lange afhankelijkheden
Trainingsefficiëntie	Kan moeilijk trainen bij lange sequenties (vanishing gradients)	Sneller dan LSTM door minder parameters	Langzamer dan GRU, maar kan betere prestaties leveren bij complexe taken
Geheugen	Kan niet goed lange-afhankelijke informatie onthouden	Beter in het onthouden van lange-afhankelijke informatie dan RNN	Uitstekend in het onthouden van lange-afhankelijke informatie
Gebruik	Geschikt voor kortere sequenties	Geschikt voor lange sequenties met minder complexiteit dan LSTM	Geschikt voor lange sequenties met complexe afhankelijkheden

Samenvatting:

- **RNNs** zijn eenvoudig, maar hebben moeite om lange-afhankelijke informatie vast te houden vanwege vanishing gradients.

- **GRUs** zijn eenvoudiger dan LSTMs en hebben minder parameters, maar zijn toch krachtig in het behouden van lange-afhankelijke informatie.
- **LSTMs** hebben een complexere architectuur met drie gates, waardoor ze in staat zijn om lange-afhankelijke informatie effectief vast te houden en beter presteren in complexere taken.

Kies het model afhankelijk van de taak:

- **RNN** voor eenvoudige taken met korte sequenties.
- **GRU** voor efficiënte training met lange sequenties.
- **LSTM** voor taken waarbij het belangrijk is om lange termijn afhankelijkheden vast te houden. 3.6 - Wat de functies van een gate zijn (remember, forget, something in between) 3.7 - Hoe een gate werkt (met een hadamard product)

Hoe werkt een gate (met een Hadamard product)?

In het kader van **GRU** (Gated Recurrent Unit) en **LSTM** (Long Short-Term Memory) netwerken spelen **gates** een cruciale rol in het reguleren van hoeveel informatie wordt doorgegeven of vergeten op basis van de huidige input en de vorige toestand. De werking van een gate kan vaak worden beschreven met behulp van een **Hadamard product** (elementgewijze vermenigvuldiging).

Wat is het Hadamard product?

Het **Hadamard product** (ook wel elementgewijze vermenigvuldiging genoemd) is de vermenigvuldiging van twee matrices van gelijke afmetingen, waarbij elk element van de eerste matrix wordt vermenigvuldigd met het overeenkomstige element van de tweede matrix.

Als we bijvoorbeeld twee vectoren $\mathbf{a} = [a_1, a_2, \dots, a_n]$ en $\mathbf{b} = [b_1, b_2, \dots, b_n]$ hebben, dan is hun Hadamard product $\mathbf{c} = \mathbf{a} \circ \mathbf{b}$, waarbij:

$$\mathbf{c} = [a_1 \cdot b_1, a_2 \cdot b_2, \dots, a_n \cdot b_n]$$

Het Hadamard product wordt vaak aangeduid met het symbool $\circ$.

Hoe werken gates in GRU en LSTM met het Hadamard product?

In **GRU** en **LSTM** werken de **gates** door de stroom van informatie te reguleren, en dit wordt vaak gedaan met behulp van een combinatie van **sigmoid**- en **tanh**-activeringsfuncties in combinatie met het Hadamard product.

1. **GRU:**

In een **GRU** zijn er twee belangrijkste gates:

- **Update gate (z)**: Regelt hoeveel van de vorige toestand moet worden behouden.
- **Reset gate (r)**: Regelt hoeveel van de vorige toestand moet worden vergeten.

Werking van de gates in GRU:

- De **update gate (z)** bepaalt hoeveel van de vorige toestand (h_{t-1}) wordt meegenomen in de nieuwe toestand (h_t). Dit gebeurt met behulp van een **Hadamard product**:

[h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h_t}]

waarbij:

- (h_{t-1}) de vorige toestand is,
- (\tilde{h_t}) de kandidaat-toestand is, die wordt berekend met behulp van de reset gate (r_t) en de huidige input (x_t),
- (z_t) de update gate is die bepaalt hoeveel van de vorige toestand moet worden behouden (door (z_t) te vermenigvuldigen met (\tilde{h_t})) en hoeveel nieuwe informatie wordt toegevoegd.

Het Hadamard product komt hier in de vorm van de multiplicatie tussen ((1 - z_t)) en (h_{t-1}), en tussen (z_t) en (\tilde{h_t}), wat zorgt voor de juiste gewichtsverhouding van oude en nieuwe informatie.

2. **LSTM:**

In een **LSTM** hebben we drie belangrijke gates:

- **Forget gate (f)**: Regelt hoeveel van de vorige toestand moet worden vergeten.
- **Input gate (i)**: Regelt hoeveel nieuwe informatie wordt toegevoegd aan de cellen.
- **Output gate (o)**: Regelt hoeveel informatie uit de cellen naar de output wordt doorgegeven.

Werking van de gates in LSTM:

LSTM's gebruiken ook het Hadamard product om de informatie te reguleren. De formules voor de update van de cellen en toestand zijn als volgt:

- **Forget gate (f)**: Het bepaalt hoeveel van de vorige cell state (C_{t-1}) moet worden vergeten:

[f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)]

- **Input gate (i)**: Het bepaalt hoeveel van de nieuwe informatie (gebaseerd op de huidige input en de vorige toestand) wordt toegevoegd:

[i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)]

- **Nieuwe kandidaat-toestand ((\tilde{C_t}))**: Dit is de nieuwe informatie die aan de cellen moet worden toegevoegd:

[\tilde{C_t} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)]

- **Celstate update (C_t)**: De nieuwe celstaat wordt berekend door de forget gate, input gate en kandidaat-toestand te combineren:

[C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C_t}]

- **Output gate (o)**: Het bepaalt hoeveel van de cellstate (C_t) wordt doorgestuurd naar de output (h_t):

[h_t = o_t \circ \tanh(C_t)]

Hier komt het Hadamard product naar voren bij de combinaties van de gates met de cellstate en de kandidaat-toestand:

- Het **Hadamard product** wordt gebruikt om te bepalen hoeveel van de vorige celstaat (C_{t-1}) moet worden behouden (door (f_t) te vermenigvuldigen met (C_{t-1})) en hoeveel van de nieuwe celstaat (\tilde{C_t}) moet worden toegevoegd (door (i_t) te vermenigvuldigen met (\tilde{C_t})).

Samenvatting:

- **Hadamard product** in **gates**: In zowel **GRU** als **LSTM** wordt het Hadamard product gebruikt om de mate van informatie-overdracht te reguleren. Het combineert verschillende componenten van de informatie (zoals de vorige toestand en de kandidaat-toestand) door ze elementgewijs te vermenigvuldigen met de output van de gates.
- **GRU** heeft minder gates (twee), waardoor het eenvoudiger is dan **LSTM**, dat drie gates heeft voor complexere controle over de informatiestroom.

Door deze mechanismen kunnen **GRU** en **LSTM** netwerken effectief lange-termijn afhankelijkheden leren en het vanishing gradient probleem aanpakken.

Waarom twee activaties in LSTM?

Sigmoid wordt gebruikt voor de gates omdat het waarden tussen 0 en 1 oplevert, waardoor het perfect geschikt is voor beslissingen over hoeveel informatie doorgegeven of vergeten moet worden. **Tanh** wordt gebruikt voor het transformeren van de celtoestand, omdat het helpt om de waarde tussen -1 en 1 te schalen, wat gunstig is voor het bewaren van de interne toestand van het geheugen in de LSTM. Door deze twee activaties te combineren, kunnen LSTM's zowel het vergeten van irrelevante informatie als het behoud van belangrijke informatie voor langere perioden mogelijk maken, wat essentieel is voor het leren van lange-afhankelijke relaties in sequentiële gegevens.

3.8 - wat de voor en nadelen van een LSTM vs GRU vs RNN zijn 3.9 - hoe windowing werkt bij timeseries, en waarom dat relevant is 3.10 - hoe 1D convolutions werken bij timeseries: 1D-convoluties (1D convolutions) worden vaak toegepast bij tijdreeksen (time series) en sequentiële gegevens (zoals spraak, tekst, sensorgegevens, enz.) om belangrijke kenmerken te extraheren zonder de noodzaak voor complexe recurrente netwerken zoals RNN's, GRU's of LSTM's. Een 1D-convolutie werkt door een filter (kern) door de inputtijdreeks te schuiven en op elk punt een convolutie uit te voeren om nieuwe representaties van de tijdreeks te creëren. Dit proces helpt bij het detecteren van patronen in de gegevens, zoals pieken, dalen, periodiciteit en trends.

Hoe werken 1D-convoluties bij tijdreeksen? Een 1D-convolutie past een filter toe op een lokale regio van de tijdreeks, wat betekent dat de filter over een klein gedeelte van de reeks schuift en een nieuwe waarde genereert door de gewogen som van de waarden in dat gedeelte te berekenen.

3.11 - begrijpt hoe een naive model werkt en wat de motivatie hierachter is (MASE metric): Een naïef model in tijdreeksanalyse is een eenvoudig model dat vaak wordt gebruikt als basislijn voor het evalueren van de prestaties van complexere modellen. Het is gebaseerd op het idee dat de toekomstige waarden van een tijdreeks op de een of andere manier direct gerelateerd zijn aan de waarden van de tijdreeks op het vorige tijdstip. Het naïeve model maakt geen gebruik van ingewikkelde voorspellende technieken of patronen, maar gebruikt in plaats daarvan een eenvoudige regel.

MASE (Mean Absolute Scaled Error) Metric

De **Mean Absolute Scaled Error (MASE)** is een veelgebruikte evaluatiemetric die de prestaties van een voorspellingsmodel vergelijkt met die van een naïef model. Het helpt bij het beoordelen van de effectiviteit van een model door het gemiddelde absolute fout te schalen op basis van de naïeve voorspelling.

De MASE wordt berekend als:

[\text{MASE} = \frac{\frac{1}{n} \sum_{t=1}^{n} |y_t - \hat{y}*t|}{\frac{1}{n-1} \sum*{t=2}^{n} |y_t - y_{t-1}|}]

waar:

- (y_t) de werkelijke waarde is op tijdstip (t),
- (\hat{y}_t) de voorspelde waarde is op tijdstip (t),
- (n) het aantal tijdsperioden is,
- de noemer is de gemiddelde absolute fout van het naïeve model.

Interpretatie van MASE:

- **MASE < 1**: Het model presteert beter dan het naïeve model.
- **MASE = 1**: Het model presteert even goed als het naïeve model.
- **MASE > 1**: Het model presteert slechter dan het naïeve model.

De MASE is schaal-invariant, wat betekent dat het niet gevoelig is voor de grootte van de tijdsreeks, waardoor het een handige metric is voor het vergelijken van modellen op verschillende datasets.

Samenvatting

- Een **naïef model** is een eenvoudig voorspellingsmodel dat vaak gebruikt wordt als benchmark voor meer complexe modellen.
- De **MASE metric** vergelijkt de prestaties van een model met een naïef model en geeft inzicht in de relatieve effectiviteit van het voorspellingsmodel.

Les 4

4.1 - wat een wordembedding is en wat de motivatie is tov one-hot-encoding 4.2 - wat de curse of dimensionality is (bv met betrekking tot de searchspace) 4.3 - wat de voordelen van Ray zijn 4.4 - Hoe bayesian search en hyperband werken 4.5 - wat een learning rate scheduler is, en hoe je kunt herkennen dat je er een nodig hebt. scheduler_kwargs={"factor": 0.5, "patience": 5} factor: Dit is de factor waarmee de leersnelheid wordt verminderd (of soms verhoogd) als er geen verbetering optreedt in de prestaties van het model na een bepaald aantal stappen (dit aantal wordt bepaald door patience). Bijvoorbeeld, als de factor 0.5 is, betekent dit dat de leersnelheid met 50% wordt verlaagd wanneer de voorwaarden worden voldaan.

patience: Dit is het aantal epochs (of iteraties) waarop het model geen verbetering mag laten zien in de prestatie (bijvoorbeeld validatieverlies) voordat de leersnelheid wordt aangepast. Als patience is ingesteld op 5, betekent dit dat de scheduler de leersnelheid pas zal verlagen nadat er 5 opeenvolgende epochs zonder verbetering zijn. 4.6 - Kent verschillende soorten schedulers (cosine warm up, reduce on plateau) en weet wanneer ze te gebruiken **StepLR**: De learning rate wordt met een vaste factor verlaagd na een bepaald aantal epochs. Wanneer te gebruiken: Dit is een eenvoudige en vaak gebruikte scheduler wanneer je weet dat de leersnelheid periodiek moet worden aangepast, bijvoorbeeld als het model snel convergeert in het begin,

maar later trager moet leren. Parameters: *step_size*: Het aantal epochs na welke de leersnelheid wordt aangepast. *gamma*: De factor waarmee de leersnelheid wordt verminderd. ExponentialLR: Goed voor simpele en regelmatige aanpassingen van de learning rate. ReduceLROnPlateau: De learning rate wordt aangepast wanneer een specifieke metriek (meestal validatieverlies of nauwkeurigheid) niet meer verbetert na een bepaald aantal epochs (patience). Dit is handig als je niet zeker weet wanneer het model stabiliseert of als het model plateau bereikt in de prestaties. Dit kan bijvoorbeeld worden gebruikt in gevallen waar je te maken hebt met onvoorspelbare convergentie. CosineAnnealingLR : Geschikt voor geleidelijke afname van de learning rate en modellen die baat hebben bij deze aanpak (bijvoorbeeld transfer learning). De learning rate wordt aangepast volgens een cosinusfunctie, wat resulteert in een geleidelijke afname van de leersnelheid van een maximum naar een minimum. Dit kan helpen om de laatste stappen van de training efficiënter te maken, door de leersnelheid langzaam te verlagen naar het einde van de training. **Cosine Warmup**: combineert een "warmup"-fase waarbij de leersnelheid langzaam toeneemt (meestal volgens een lineaire of cosinuscurve) gedurende de eerste paar epochs, gevolgd door een periodieke afname van de learning rate (meestal via een cosinusfunctie). Wanneer te gebruiken: Cosine warmup is vaak effectief voor taken waarbij een lage initiële leersnelheid gewenst is om "pre-train" stabiliteit te garanderen (bijvoorbeeld bij transfer learning). Het kan ook nuttig zijn voor bepaalde generative modellen zoals GAN's. Cyclical Learning Rate (CLR): Geweldig als je wilt experimenteren met verschillende leersnelheden en flexibelere training zoekt. OneCycleLR: Aanbevolen voor snelle convergentie en daarna nauwkeurige afstemming van het model. 4.7 - Begrijpt in welke situaties transfer-learning zinvol is: Transfer learning is bijzonder waardevol in situaties waarin je met beperkte data werkt of wanneer je het model sneller wilt laten convergeren door gebruik te maken van reeds verworven kennis uit een gerelateerde taak Transfer learning is bijzonder nuttig wanneer:

- Je beperkte data hebt voor de taak die je wilt oplossen.
- Je tijd of middelen wilt besparen door een model dat al is getraind op grote hoeveelheden gegevens opnieuw te gebruiken.
- Je wilt profiteren van reeds geleerde representaties voor een nieuwe, verwante taak.
- Je met complexe taken werkt die moeilijk vanaf nul te trainen zijn.

De student kan: 4.8 - de parameters in een pretrained model fixeren zodat het uitgebreid en gefinetuned kan worden 4.9 - Een pretrained model uitbreiden met een extra neuraal netwerk 4.10 - Een python script bouwen dat een configureerbaar model via Ray hypertuned. 4.11 - De student kan redeneren over de grootte van de hyperparameter ruimte, daar afwegingen in maken (curse of dimensionality) en prioriteiten stellen in de tuning van hyperparameters zoals lossfuncties, learning rate, units, aantal lagen, aantal filters, combinatie van Dense / Convolution. 4.12 - een afweging maken tussen de verschillende manieren om een model te monitoren (gin, tensorboard, mlflow, ray)

LES 5

5.1 - Precision vs Recall trade off, Confusion matrix, ethische problemen van de trade off

Precision vs Recall Trade-off

In machine learning en informatie-extractie, vooral bij classificatietaken, worden **precision** en **recall** vaak gebruikt als evaluatiemetrics. Deze twee metrics zijn essentieel voor het beoordelen van de prestaties van een model, maar ze zijn vaak in conflict met elkaar. Het **trade-off** tussen precision en recall betekent dat een verbetering in de ene metric meestal leidt tot een verslechtering in de andere.

Definitie van Precision en Recall

1. **Precision**: Precision meet hoe nauwkeurig de positieve voorspellingen van het model zijn. Het geeft aan hoeveel van de voorspelde positieve gevallen daadwerkelijk correct zijn.

   [\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}] Waar:

   - **True Positives (TP)**: Correct voorspelde positieve gevallen.
   - **False Positives (FP)**: Onterecht voorspelde positieve gevallen.

2. **Recall (Sensitiviteit of True Positive Rate)**: Recall meet hoe goed het model daadwerkelijk alle positieve gevallen kan identificeren. Het geeft aan hoeveel van de werkelijke positieve gevallen correct werden voorspeld.

   [\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}] Waar:

   - **True Positives (TP)**: Correct voorspelde positieve gevallen.
   - **False Negatives (FN)**: Positieve gevallen die onterecht als negatief werden voorspeld.

Het Trade-off Tussen Precision en Recall

De **trade-off** ontstaat doordat het verhogen van één metric vaak leidt tot het verlagen van de andere.

- **Verhogen van Precision**: Als je de drempel voor het voorspellen van een positief geval verhoogt (bijvoorbeeld door een model alleen positieve gevallen te laten voorspellen wanneer het heel zeker is), krijg je een hogere precision. Dit kan echter leiden tot een lagere recall, omdat sommige werkelijke positieve gevallen worden gemist (False Negatives).

- **Verhogen van Recall**: Als je de drempel verlaagt om meer gevallen als positief te classificeren (bijvoorbeeld door een model te laten voorspellen dat iets positief is, zelfs als het niet helemaal zeker is), zal de recall stijgen. Dit kan echter de precision verlagen, omdat er meer False Positives zullen zijn.

Voorbeeld

Stel je voor dat je een model hebt dat kanker detecteert:

- **Precision hoog**: Het model voorspelt alleen "kanker" als het zeer zeker is. Dit betekent dat wanneer het zegt "kanker", het bijna altijd juist is, maar mogelijk mis je enkele echte gevallen van kanker (lage recall).

- **Recall hoog**: Het model voorspelt vaker "kanker", waardoor het meer echte gevallen van kanker oppikt. Maar het kan ook onterecht veel mensen "kanker" voorspellen, die het niet hebben (lage precision).

F1-Score

De **F1-score** is een metriek die een balans probeert te vinden tussen precision en recall. Het is het harmonisch gemiddelde van de twee:

[F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}]

De F1-score is handig wanneer zowel precision als recall belangrijk zijn en je een enkele metric wilt die het compromis tussen de twee weerspiegelt.

Interpretatie van de F1-score:

- **Hoge F1-score**: Het model heeft zowel een goede precision als recall.
- **Lage F1-score**: Het model presteert slecht op een van beide metrics (of beide).

Keuze tussen Precision en Recall

De keuze tussen het optimaliseren van precision of recall hangt af van de specifieke context van de taak:

- **Als je False Positives belangrijker vindt** (bijvoorbeeld bij e-mailspamdetectie, waar je geen onterecht belangrijke e-mail als spam wilt markeren), dan kan je de voorkeur geven aan **precision**.
- **Als je False Negatives belangrijker vindt** (bijvoorbeeld bij medische diagnoses, waar je zeker wilt zijn dat je geen ziekten mist), dan zou je meer nadruk moeten leggen op **recall**.

Samenvatting

- **Precision** is de nauwkeurigheid van de *positieve voorspellingen*.
- **Recall** is het vermogen om alle werkelijke positieve gevallen te vinden.
- Er is vaak een **trade-off** tussen precision en recall, waarbij het verbeteren van de ene metric ten koste van de andere gaat.
- De **F1-score** biedt een gebalanceerde evaluatie van de prestaties, vooral wanneer zowel precision als recall belangrijk zijn.

Het kiezen van de juiste benadering hangt af van de specifieke vereisten van de applicatie.

5.2 - de motivatie achter attention (vs convolutions)

Motivatie Achter Attention (vs Convolutions)

In recente jaren heeft **Attention** veel aandacht gekregen, vooral in de context van neurale netwerken en natuurlijke taalverwerking (NLP). Het concept van attention heeft verschillende voordelen in vergelijking met traditionele technieken zoals **convoluties** in **Convolutional Neural Networks (CNN's)**. In dit document onderzoeken we de motivatie en de belangrijkste verschillen tussen **attention** en **convoluties**.

Wat zijn Convoluties?

Convoluties zijn de kern van **Convolutional Neural Networks (CNN's)**, die oorspronkelijk populair werden in beeldherkenningstaken. Het basisidee van convoluties is om lokale patronen in de invoerdata (zoals pixels in een afbeelding) te detecteren door middel van een filter (kern). Deze filter beweegt over de input en berekent de convolutie, wat resulteert in feature maps die lokale kenmerken vastleggen.

- **Voordeel van convoluties**: Convoluties zijn effectief voor het detecteren van lokale patronen zoals randen, hoeken en texturen in afbeeldingen. Ze zijn verticaal, horizontaal en in meerdere schalen toepasbaar, wat ze krachtig maakt voor visuele taken.
- **Beperkingen van convoluties**: Convoluties hebben echter moeite om lange-afstand afhankelijkheden vast te leggen. In een afbeelding bijvoorbeeld, is er weinig context buiten het lokale gebied van een filter. Dit kan problematisch zijn bij complexere taken, zoals tekstverwerking en lange sequenties van data.

Wat is Attention?

Attention is een mechanismen dat oorspronkelijk werd voorgesteld in de context van machinevertaling, maar nu wijdverbreid wordt gebruikt in NLP en andere domeinen zoals vision. Het idee van attention is om dynamisch te bepalen welke delen van de input op een bepaald moment het meest relevant zijn voor het maken van een voorspelling. Dit staat modellen toe om verschillende delen van de invoer op verschillende tijdstippen meer "aandacht" te geven.

Een bekend voorbeeld van attention is de **self-attention** in Transformer-modellen, zoals **BERT** en **GPT**.

Hoe werkt Attention?

Het principe van attention is eenvoudig:

- Voor een bepaalde invoer (bijvoorbeeld een woord in een zin), berekent het model een gewicht voor elk ander woord in de zin (of elke andere input).
- Deze gewichten reflecteren hoeveel **aandacht** elk woord verdient bij het voorspellen van de huidige output.

Bijvoorbeeld, in een zin als "De kat zat op de mat", kan het model meer aandacht geven aan "kat" als het woord "zat" moet worden voorspeld, en minder aan andere woorden zoals "de" of "mat".

Types van Attention:

- **Self-attention**: Elke positie in de input kijkt naar alle andere posities om context te verzamelen (zoals in Transformers).
- **Bahdanau Attention**: Gebruikt gewichten die de relevantie van verschillende invoer-elementen bepalen in een sequence-to-sequence taak.

Vergelijking: Attention vs Convoluties

1. **Lange-afstandsafhankelijkheden**:

- **Convoluties**: Convoluties kunnen alleen lokale patronen herkennen. Als een object of patroon zich ver van andere objecten bevindt, kunnen convoluties moeite hebben om deze lange-afstandsafhankelijkheden vast te leggen.
- **Attention**: Attention maakt het mogelijk om **globale afhankelijkheden** vast te leggen, ongeacht de afstand tussen de relevante elementen in de input. Dit is vooral nuttig voor taken zoals taalverwerking, waar de betekenis van een woord vaak afhankelijk is van de context van woorden verderop in de zin.

2. **Flexibiliteit**:

- **Convoluties**: Het aantal filterparameters is meestal vast en wordt vooraf gedefinieerd. Hierdoor is het moeilijk om het model dynamisch aan te passen aan verschillende taken zonder de architectuur te veranderen.
- **Attention**: Het model past zich dynamisch aan door de nadruk te leggen op verschillende delen van de invoer. Dit maakt het model flexibeler in vergelijking met convoluties, vooral voor sequentiële data en taken zoals vertaling en tekstbegrip.

3. **Computational Efficiency**:

- **Convoluties**: Convoluties zijn relatief **computationally efficiënt**, omdat ze lokale patronen parallel kunnen verwerken. Dit is een reden waarom CNN's goed presteren bij beeldherkenning.
- **Attention**: **Self-attention** in zijn klassieke vorm, zoals in het Transformer-model, kan computationeel intensiever zijn, vooral bij lange sequenties. Dit komt doordat elk element in de sequentie met elk ander element interactie heeft, wat leidt tot een kwadratische complexiteit ($O(n^2)$).

4. **Verwerking van Sequentiële Data**:

- **Convoluties**: CNN's zijn goed in het verwerken van **local dependencies**, maar missen de mogelijkheid om lange-afstandsafhankelijkheden efficiënt te modelleren.
- **Attention**: Attention is ontworpen om sequentiële gegevens zoals tekst, waar de volgorde en de relaties tussen de elementen essentieel zijn, beter te verwerken. Het biedt de mogelijkheid om een woord in een zin contextueel te begrijpen, ongeacht de positie ervan in de sequentie.

5. **Interpretabiliteit**:

- **Convoluties**: De interpretatie van de filters in CNN's is vaak moeilijker, omdat ze vaak op een abstract niveau werken om patronen te detecteren.
- **Attention**: Attention heeft een grotere **interpretabiliteit**, omdat het expliciet gewichten geeft aan verschillende delen van de invoer. Dit maakt het makkelijker te begrijpen welk deel van de input het model "belangrijk" vindt voor een specifieke voorspelling.

Motivatie Achter Attention

- **Contextual Awareness**: In veel toepassingen, zoals taalmodellen, is de volgorde en de onderlinge afhankelijkheid van elementen cruciaal. Attention biedt een krachtig mechanisme om dynamisch de context te begrijpen, terwijl convoluties moeite hebben om lange-afstandsrelaties vast te leggen.

- **Flexibiliteit en Aanpasbaarheid**: Attention kan zich gemakkelijker aanpassen aan verschillende datadomeinen (zoals tekst en afbeeldingen) door de nadruk te leggen op relevante delen van de invoer.

- **Parallelisatie**: Moderne varianten van attention, zoals de **Transformer**, maken efficiënte parallelle verwerking mogelijk, wat een grote versnelling biedt in training en inferentie, zelfs bij lange sequenties.

Conclusie

- **Convoluties** blijven effectief voor taken die voornamelijk afhangen van lokale patronen (zoals in beeldherkenning), maar hebben moeite met het modelleren van lange-afstandsafhankelijkheden.
- **Attention** biedt een krachtiger alternatief voor taken die complexe, lange-afstandsrelaties vereisen, zoals taalverwerking en sequentiële data. Het biedt **flexibiliteit** en de mogelijkheid om zowel lokale als globale contexten vast te leggen.

5.3 - wat een semantische vectorruimte is

Wat is een Semantische Vectorruimte?

Een **semantische vectorruimte** is een wiskundige representatie van betekenis, waarbij woorden, zinnen, of andere taalobjecten worden omgezet in vectoren in een hoge-dimensionale ruimte. In deze ruimte wordt de

betekenis van taalobjecten vastgelegd door hun **relaties** en **semantische overeenkomsten** met andere objecten in de ruimte.

Het idee is om betekenis niet alleen op basis van de letters of symbolen van een woord vast te leggen, maar door de **context** en **relaties** tussen woorden, zinnen, of zelfs documenten te begrijpen. De semantische vectorruimte biedt een manier om **taal op een numerieke manier** te representeren, zodat computers gemakkelijker met natuurlijke taal kunnen werken.

Kernconcepten van Semantische Vectorruimtes

1. **Vectorrepresentatie van Taal**:
    - In een semantische vectorruimte worden woorden, zinnen, of zelfs grotere taalconstructies zoals documenten gerepresenteerd door **vectoren** (numerieke lijsten van getallen). Deze vectoren bevinden zich in een hoge-dimensionale ruimte, vaak met honderden of duizenden dimensies.
    - Woorden die semantisch dicht bij elkaar liggen (d.w.z. die in een vergelijkbare context voorkomen of een vergelijkbare betekenis hebben) worden gerepresenteerd door vectoren die **dicht bij elkaar** liggen in de ruimte.

2. **Relaties en Semantische Betekenis**:
    - In een semantische vectorruimte weerspiegelen de **relaties tussen woorden** de betekenis van die woorden in context. Bijvoorbeeld, in zo'n ruimte zullen de woorden "koning" en "koningin" relatief dicht bij elkaar liggen, omdat ze vaak in vergelijkbare contexten worden gebruikt.
    - De vectoren kunnen zelfs **verhouding** tussen woorden weergeven, zoals het voorbeeld: [\text{"koning" - "man" + "vrouw" = "koningin"}]
    - Dit toont aan hoe semantische vectorruimtes niet alleen de gelijkenis tussen woorden kunnen vangen, maar ook **relaties** en **verschillen** tussen woorden kunnen modelleren.

3. **Gebruik van Embeddings**:
    - **Word embeddings** zoals **Word2Vec**, **GloVe**, en **FastText** zijn populaire technieken die woorden omzetten naar vectoren in een semantische vectorruimte. Deze embeddings worden getraind op grote hoeveelheden tekst, zodat de vectoren de semantische betekenis van woorden reflecteren.
    - **Sentence embeddings** of **document embeddings** zijn uitbreidingen van dit idee, waarbij niet alleen woorden, maar ook langere tekstfragmenten een vectorrepresentatie krijgen.

Toepassingen van Semantische Vectorruimtes

1. **Zoeken en Informatie Retrieval**:
    - Semantische vectorruimtes kunnen worden gebruikt voor **zoekmachines**, waarbij de zoekquery en de documenten in de vectorruimte worden geplaatst. Documenten die semantisch dicht bij de zoekopdracht liggen, worden als relevanter beschouwd, zelfs als ze niet exact dezelfde woorden bevatten.

2. **Tekstanalyse en Sentimentanalyse**:
    - Vectorruimtes worden gebruikt in veel NLP-taken zoals **sentimentanalyse** of **topicmodellering** om te begrijpen of een tekst positief, negatief of neutraal is, en om te detecteren welke

onderwerpen aanwezig zijn in een tekst.

3. **Vertaling en Meertalige Modellen**:
    - In **machinevertaling** worden semantische vectorruimtes gebruikt om woorden in verschillende talen te verbinden. Een goed getraind semantisch model kan bijvoorbeeld "apple" in het Engels en "manzana" in het Spaans dichter bij elkaar plaatsen in de vectorruimte, omdat ze dezelfde betekenis vertegenwoordigen.
4. **Synoniemen en Clustering**:
    - Woorden die semantisch verwant zijn, zoals "auto" en "voertuig", zullen dicht bij elkaar liggen in de semantische vectorruimte. Dit maakt het mogelijk om synoniemen te detecteren en woorden te clusteren op basis van hun betekenis.

Voordelen van Semantische Vectorruimtes

1. **Contextualisatie van Betekenis**:
    - Semantische vectorruimtes kunnen **contextualisatie** van betekenis vastleggen. Dit betekent dat hetzelfde woord in verschillende contexten verschillende vectoren kan hebben, wat de precisie van taalmodellen verhoogt.
2. **Verbeterde Zoekfunctionaliteit**:
    - Omdat de betekenis van woorden en zinnen in vectorruimtes is vastgelegd, kunnen zoeksystemen **semantisch begrijpen** wat de gebruiker bedoelt, niet alleen wat de exacte zoekwoorden zijn.
3. **Schaalbaarheid**:
    - Semantische vectorruimtes kunnen worden getraind op enorme hoeveelheden tekst en kunnen gemakkelijk worden toegepast op verschillende talen en domeinen.

Conclusie

Een **semantische vectorruimte** is een krachtige manier om taal te representeren door woorden, zinnen of documenten als vectoren in een hoge-dimensionale ruimte te plaatsen. Deze representaties helpen om **betekenis** en **relaties** tussen taalobjecten vast te leggen, wat nuttig is voor veel toepassingen in natuurlijke taalverwerking (NLP), zoals zoekopdrachten, vertaling, tekstclassificatie en meer. Door de krachtige relaties en semantische contexten die worden vastgelegd, biedt de semantische vectorruimte een geavanceerde manier om taal te begrijpen en te verwerken.

5.4 - Hoe het "reweighing" van word-embeddings werkt met behulp van attention "Reweighing" van word-embeddings met behulp van attention is een proces waarbij de representatie van woorden in een tekst wordt aangepast op basis van de context, met behulp van een attention-mechanisme. Dit mechanisme zorgt ervoor dat het model dynamisch kan bepalen welke woorden in de context belangrijker zijn voor de betekenis van een specifiek woord. Stappen:

1. Initialisatie van woord-embeddings: Een woord wordt eerst omgezet in een embedding (meestal een vector in een hoge-dimensionale ruimte) via een vooraf getraind model zoals Word2Vec, GloVe of de

woordrepresentaties die in een transformer-gebaseerd model zoals BERT worden geleerd. Deze embeddings bevatten initieel de betekenis van woorden, maar missen vaak context.

2. Toepassen attention-mechanisme: Het attention-mechanisme (bijvoorbeeld in transformer-modellen zoals BERT of GPT) werkt door een gewicht toe te kennen aan elk woord in de context van het doelwoord (het woord waar we onze focus op willen richten). Dit wordt gedaan door twee belangrijke stappen:

- Query, Key en Value: Elke woordembedding wordt omgezet in drie vectoren: een query (Q), een key (K) en een value (V). Dit gebeurt door de *oorspronkelijke embeddings te vermenigvuldigen met getrainde gewichtsmatrices*.

- Calculatie van attention-gewichten: De aandacht wordt berekend door de vergelijkbaarheid tussen de query van het doelwoord en de keys van de andere woorden in de context te meten, meestal door een dot-product en daarna een softmax-functie toe te passen. Het resultaat is een set van gewichten die aangeven hoeveel invloed elk woord in de context heeft op het doelwoord.

3. Reweighing van de woord-embeddings: De oorspronkelijke embeddings van de woorden in de context worden vervolgens gewogen door de berekende aandacht-gewichten. Dit betekent dat de vectoren van de contextwoorden die meer relevant zijn voor de betekenis van het doelwoord een grotere weging krijgen, terwijl woorden die minder relevant zijn een kleinere invloed hebben. Dit stelt het model in staat om de betekenis van een woord dynamisch aan te passen op basis van de omliggende woorden.

5.5 - waarom scaling en masking nodig zijn Het **scaling**-proces is belangrijk om te voorkomen dat de waarden van de aandacht te groot of te klein worden, wat kan leiden tot numerieke instabiliteit of inefficiëntie in het leren.

Masking is een techniek die wordt gebruikt om ervoor te zorgen dat bepaalde woorden in de input geen invloed hebben op andere woorden in bepaalde gevallen, bijvoorbeeld in sequentiële taken of bij het verwerken van een "future" context. De aandacht (Attention) kan wiskundig worden uitgedrukt als:

$$ \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V $$ Hier zorgt $${\sqrt{d_k}} $$ ervoor dat de schaal van de dot-producten niet te groot wordt.

waarbij:

- (Q) de query-matrix is,
- (K) de key-matrix is,
- (V) de value-matrix is,
- (d_k) de dimensie van de key-vectoren is.
-

5.6 - wat multihead attention is Multihead Attention is een concept dat wordt gebruikt in moderne transformer-architecturen, zoals die in BERT, GPT, en andere geavanceerde modellen voor natuurlijke taalverwerking en vision-taken. Het is een essentieel onderdeel van de self-attention-mechanisme, en helpt modellen om meerdere representaties van de data tegelijk te verwerken door verschillende aandachtshoofden parallel te laten werken.

Wat is Attention? Voordat we ingaan op multihead attention, is het handig om te begrijpen wat attention is.

In de context van neural networks verwijst attention naar het mechanisme waarbij het model bepaalt welke delen van de input belangrijker zijn voor een specifieke taak, en focus aanbrengt op die delen. Dit betekent dat het model dynamisch kan beslissen waar het zijn "aandacht" op moet richten binnen de input.

In self-attention (de basis voor transformer-architecturen) krijgt elke positie in de input de kans om informatie van andere posities in de sequentie te verzamelen, afhankelijk van hun relatieve relevantie.

Wat is Multihead Attention? Multihead Attention breidt dit concept van attention verder uit door meerdere "hoofden" van aandacht parallel te gebruiken. Elk hoofd is in staat om verschillende aandachtspatronen te leren en zo diverse aspecten van de input te begrijpen. In plaats van één enkele aandachtsoverweging, leren meerdere hoofden tegelijkertijd verschillende aspecten van de relatie tussen tokens. Dit zorgt ervoor dat het model in staat is om complexere en veelzijdigere representaties van de input te leren. Stel je voor dat je een zin hebt als input. In plaats van één enkele aandachtsscore voor elk paar van woorden in de zin, leert multihead attention verschillende, parallelle sets van aandachtspatronen. Elk hoofd kan zich richten op verschillende aspecten van de zin:

Eén hoofd zou bijvoorbeeld kunnen leren welke woorden grammaticaal belangrijk zijn. Een ander hoofd zou zich kunnen concentreren op semantische relaties tussen woorden. Een ander hoofd zou mogelijk focussen op afhankelijkheden over lange afstanden (bijvoorbeeld, het verband tussen het onderwerp van een zin en een werkwoord). Na het berekenen van deze verschillende aandachtspatronen voor elk hoofd, worden de resultaten samengevoegd om een rijkere en meer veelzijdige representatie van de invoer te krijgen.

5.7 - wat positional encoding is, en waarom dat behulpzaam kan zijn **Positional Encoding** voegt volgorde-informatie toe aan de tokenrepresentaties in een transformer-model. Dit is nodig omdat transformers geen inherent begrip van volgorde hebben (in tegenstelling tot RNN's en LSTM's). De meest gebruikte techniek voor positional encoding is de sinus-cosinusfunctie, die het mogelijk maakt om verschillende posities effectief te representeren. Positional encoding helpt transformers om de volgorde van tokens in een sequentie te begrijpen, zonder de noodzaak voor recursieve netwerken, en ondersteunt parallelle verwerking van sequentiële data.

5.8 - kan uitleggen hoe attention behulpzaam is bij timeseries in het algemeen, en NLP in het bijzonder. In traditionele tijdreeksmodellen zoals RNN's en LSTM's kunnen modellen moeite hebben om lange-afstandsafhankelijkheden vast te leggen. Dit betekent dat als er belangrijke gebeurtenissen zijn die ver in de tijd liggen, de informatie vaak verloren gaat (bijvoorbeeld door verdwijnende gradiënten). Attention kan echter rechtstreeks de relatie tussen tijdstappen leggen, zelfs als ze ver van elkaar verwijderd zijn.In plaats van de volledige reeks gegevens in volgorde te verwerken, kan een attention-mechanisme selectief focussen op de meest relevante tijdstappen. In tijdreeksen helpt attention modellen om belangrijke tijdstappen in lange sequenties te identificeren, lange-afstandsafhankelijkheden vast te leggen en efficiënter te werken door parallel verschillende tijdstappen te analyseren.

In NLP stelt attention modellen in staat om relaties tussen woorden of zinnen, zelfs over lange afstanden, vast te leggen, wat essentieel is voor het begrijpen van de betekenis in tekst. 5.9 - kent het verschil in dimensionaliteit van tensors (2D tensors, 3D tensors, 4D tensors) voor de diverse lagen (Dense, 1D en 2D Convulutionl, MaxPool, RNN/GRU/LSTM, Attention, Activation functions, Flatten) en hoe deze met elkaar te combineren zijn.

Vaardigheden:

5.10 - een attention layer toevoegen aan een RNN model 5.11 - kan een preprocessor voor NLP maken (bv punctuation, lowercase, spaces en xml strippen) 5.12 - een datapreprocessor aanpassen voor een dataset

Les 6

De student begrijpt 6.1 - hoe de architectuur van een autoencoder in elkaar zit Een **autoencoder** is een type neuraal netwerk dat voornamelijk wordt gebruikt voor unsupervised learning, en is ontworpen om inputdata te comprimeren naar een lagere dimensionale representatie (de encoder) en vervolgens deze representatie te reconstrueren naar de originele data (de decoder). Het doel is om de belangrijkste kenmerken van de data te leren, zodat de originele data goed wordt gereconstrueerd, terwijl ruis of minder belangrijke informatie wordt weggelaten. De architectuur van een autoencoder bestaat uit drie hoofdcomponenten: de encoder, de bottleneck (de latent space) en de decoder 6.2 - Wat een latent space is, en hoe je die kunt visualiseren **latent space** bevat de gecodeerde informatie van de inputdata in een lagere dimensie. Dit is de kern van wat de autoencoder probeert te leren – een efficiënte representatie van de data. Eigenschappen: De bottleneck bevat de belangrijkste kenmerken van de data die het model heeft geleerd, en is meestal veel kleiner in dimensie dan de input zelf. Beperkingen: De bottleneck vormt een soort informatiebeperkingen voor het model. Het netwerk moet leren om de belangrijkste informatie in deze kleinere representatie vast te leggen, wat de autoencoder kan helpen bij het ontdekken van interessante patronen in de data. Het doel van een autoencoder is om de input zo goed mogelijk te reconstrueren. Daarom wordt het model getraind om de herbouwde output te vergelijken met de originele input en deze fout te minimaliseren. Lossfunctie: De meest gebruikelijke loss-functie is de mean squared error (MSE) voor continue waarden (bijvoorbeeld bij afbeeldingen), of binary cross-entropy voor binaire data. Het model wordt getraind om de verschillen tussen de originele data en de gereconstrueerde output te minimaliseren. Dit wordt vaak gedaan door een backpropagation-algoritme, waarbij de gewichten van de encoder en decoder worden aangepast om de fout te verkleinen. 6.3 - Wat het contrast is tussen autoencoders en Supervised learning op punten als de latent space en de loss functie. **Latente ruimte** is de gecomprimeerde representatie van de inputdata die door het encoder-decoder netwerk wordt geleerd. Het doel van een autoencoder is om de belangrijkste patronen in de data vast te leggen in een kleinere dimensionale ruimte. Deze latente ruimte is vaak een vector die de kernkenmerken van de inputdata vertegenwoordigt. Autoencoders proberen de input te reconstrueren uit deze gecomprimeerde latente ruimte. De latente ruimte van een autoencoder is niet gebaseerd op enige specifieke labels of doelen. Het is een ongecontroleerde representatie die puur de structurele kenmerken van de data probeert vast te leggen. Latente ruimte is ook aanwezig in veel supervised learning-modellen, maar de latente representaties worden geleerd met behulp van labels (doelen). De latente ruimte in supervised learning is ontworpen om de kenmerken van de data te leren die belangrijk zijn voor het voorspellen van de bijbehorende labels. Lossfuncties: voor autoencoders is de lossfunctie het resultaat van het verschil tussen reconstructiefout tussen de oorspronkelijke input en de gereconstrueerde output van het model. Het doel van de autoencoder is om de input te reconstrueren met zo min mogelijk fout. De meest gebruikte lossfunctie is de mean squared error (MSE) voor continue waarden of binary cross-entropy voor binaire data. De lossfunctie meet dus simpelweg hoe goed de output van de decoder de originele input benadert. In supervised learning is de lossfunctie afhankelijk van de taak. Voor classificatie kan de lossfunctie bijvoorbeeld categorical cross-entropy zijn, en voor regressie kan dit mean squared error (MSE) zijn. Het belangrijkste verschil is dat de lossfunctie in supervised learning afhankelijk is van de labels van de data.

Kenmerk	Autoencoders (Unsupervised Learning)	Supervised Learning

Kenmerk	Autoencoders (Unsupervised Learning)	Supervised Learning
Latente ruimte	Gecomprimeerde representatie van de data zonder gebruik van labels. Probeert de belangrijkste kenmerken van de inputdata vast te leggen.	Latente representaties worden geleerd om het model te helpen de labels (doelen) correct te voorspellen. Het is label-gedreven.
lossfunctie	Gebaseerd op de reconstructiefout (bijv. MSE, cross-entropy). De input zelf is het doel van de reconstructie.	Gebaseerd op de fout tussen de voorspelling en de werkelijke labels (bijv. MSE voor regressie, cross-entropy voor classificatie).
Gebruik van labels	Geen labels nodig. Het model probeert alleen de inputdata te reconstrueren.	Labels zijn noodzakelijk. Het model leert de relatie tussen input en de bijbehorende labels.

6.4 - Wat praktische toepassingen zijn van autoencoders en hoe de latent space gebruikt kan worden als half-fabrikaat voor andere modellen de encoder is goed om bijv. data samen te vatten of te analiseren, voor het bepalen van anomalies of weghalen of toevoegen van noise 6.5 - Hoe anomaly detection met een autoencoder werkt Anomaly detection met een autoencoder is een techniek waarbij het model wordt getraind om "normale" data te leren en vervolgens te beoordelen of nieuwe data afwijkt van wat het model als normaal beschouwt. Het idee is dat de autoencoder leert om de belangrijke patronen in de data te reconstrueren, en wanneer een nieuwe inputafwijking (anomalie) deze patronen niet volgt, zal de reconstructiefout groter zijn.

1. Train de autoencoder met alleen normale data.
2. Bereken de reconstructiefout voor nieuwe gegevens.
3. Vergelijk de reconstructiefout met een drempel:
4. Kleine fout → normale gegevens.
5. Grote fout → anomalie. Classificeer de anomalieën door de gegevens met grote fouten als afwijkend te markeren. 6.6 - Hoe unsupervised classification met een autoencoder werkt Stel je voor dat je een autoencoder hebt die afbeeldingen van normale objecten leert reconstrueren. Bij het testen met een afbeelding van een onbekend object (bijvoorbeeld een defect object in een fabriek) zal de autoencoder moeite hebben om deze afbeelding goed te reconstrueren. Het verschil tussen de originele afbeelding en de gereconstrueerde afbeelding (de reconstructiefout) zal groter zijn, wat de anomalie aangeeft.

6.7 - Wat de elementen van een siamese network zijn Een **Siamese netwerk** is een type neuraal netwerk dat wordt gebruikt voor het vergelijken van twee vergelijkbare inputs om te bepalen of ze dezelfde of verschillende klassen vertegenwoordigen. Het wordt voornamelijk gebruikt voor vergelijkingstaken zoals gezichtsherkenning, handtekeningverificatie, schoenmaatmatching, gebruiker-verificatie, en andere gevallen waarbij het doel is om de gelijkenis of verschillen tussen twee invoeren te meten.

1. Identieke subnetwerken: Twee netwerkcomponenten die dezelfde parameters delen.
2. Feature extractie: Elke input wordt door zijn eigen netwerk verwerkt om een latente representatie te krijgen.
3. Samenvoeging van representaties: De representaties van de twee inputs worden gecombineerd om hun gelijkenis te meten.
4. Afstandsfunctie: De gelijkenis wordt gemeten met een afstandsmaat, zoals Euclidische afstand of cosine-similariteit.

5. Verliesfunctie: De contrastieve verliesfunctie of triplet loss wordt gebruikt om de afstand tussen gelijke en ongelijke paren te optimaliseren.
6. Beslissingslaag: Na de berekening van de afstand wordt een beslissingslaag toegevoegd, meestal een sigmoid- of softmax-functie, die de kans bepaalt dat de twee invoeren bij dezelfde klasse horen (bijvoorbeeld de kans dat twee gezichten van dezelfde persoon zijn). De uitkomst is een kans of klasse-uitvoer: Bij een *sigmoid* functie (resultaat tussen 0-1) wordt het resultaat vaak geïnterpreteerd als de kans of waarschijnlijkheid dat de twee invoeren gelijk zijn (zelfde klasse). Bij *softmax* kan het systeem een classificatie uitvoeren, vooral als er meerdere klassen betrokken zijn.

6.8 - Wanneer je siamese networks zou willen gebruiken Toepassingen van Siamese Netwerken: Gezichtsherkenning: Vergelijken van twee gezichten om te bepalen of ze dezelfde persoon zijn. Signatuurverificatie: Controleren of twee handtekeningen van dezelfde persoon zijn. Beeldvergelijking: Vergelijken van twee afbeeldingen om te bepalen of ze hetzelfde object of dezelfde scène bevatten. Document Matching: Vergelijken van tekstfragmenten of documenten om te zien of ze semantisch vergelijkbaar zijn. 6.9 - Hoe je een dataloader moet aanpassen voor een autoencoder De DataLoader voor een autoencoder moet zowel de invoer als de doelen leveren, die meestal identiek zijn (de invoer wordt gereconstrueerd door het model). Je maakt een custom Dataset-klasse die ervoor zorgt dat de invoer gelijk is aan de doeloutput. Je kunt de DataLoader gebruiken om batches van gegevens te laden en aan het model te geven. Tijdens de training worden de invoer en de doeloutput gebruikt om de reconstructiefout (bijvoorbeeld MSE) te berekenen en de gewichten van het model te optimaliseren. 6.10 - Wat is de motivatie voor JEPA? Hoe werkt JEPA in vergelijking met autoencoders / genai? **JEPA** gaat ervanuit dat het neurale netwerk een hogere niveau van abstractie kan leren door een onderdeel van een foto te maskeren waardoor het gedwongen wordt om de ontbrekende onderdeel te leren. De architectuur is nog steeds een autoencoder. JEPA (Joint Embedding Predictive Architecture) is een relatief nieuwe benadering in de machine learning- en computer vision-domeinen, die is ontworpen om de representaties van de data (in de vorm van embeddings) te verbeteren, met behulp van zelfsupervisie en contrastieve leren. JEPA werd gepresenteerd als een manier om betere en robuustere representaties van data te leren zonder afhankelijk te zijn van zware supervisie of grote hoeveelheden gelabelde data. De motivatie voor JEPA komt voort uit de behoefte om effectievere en meer informatieve representaties te creëren voor downstreamtaken, zoals classificatie, detectie en segmentatie, zonder expliciete labelinformatie.

De student kan 6.11 - Een autoencoder netwerk ontwerpen en hypertunen 6.12 - Een encoder of decoder uit een getrainde autoencoder hergebruiken

Les 7: Graph Neural Networks en Graph Convulutional Networks (GCNs)

Book: https://www.cs.mcgill.ca/~wlh/grl_book/files/GRL_Book.pdf https://www.youtube.com/watch?v=0YLZXjMHA-8

Idea: graph predictions combine the information of nearby nodes into one compact layer. It works just as bij CNN, where an aggregration takes place on node levels instead of pixel level.