Package ‘pinnEV’

August 15, 2022

Type Package

Title Partially-Interpretable Neural Networks for Extreme Value modelling

Version 0.1.0

Author Jordan Richards

Maintainer The package maintainer <jordan.richards@kaust.edu.sa>

Description Methodology for fitting marginal extreme value (and associated) models using partially-interpretable neural networks. Networks are trained using the R interface to Keras with custom loss functions taken to be penalised versions of the negative log-likelihood for associated models.

License use_mit_license()

Encoding UTF-8

Depends R (>= 3.4)

Imports reticulate, keras, tfprobability, stats, evd

Roxygen list(markdown = TRUE, old_usage = TRUE)

RoxygenNote 7.2.1

R topics documented:

bGEV *The blended-GEV distribution*

Description

Distribution function, quantile function and random generation for the blended generalised extreme value (bGEV) distribution with location equal to `q_alpha`, spread equal to `s_beta` and shape equal to `xi`. Note that unlike similar functions in package `stats`, these functions accept only scalar inputs, rather than vectors, for the parameters.

Usage

```
pbGEV(y, q_alpha, s_beta, xi, alpha = 0.5, beta = 0.5, p_a = 0.05,
  p_b = 0.2, c1 = 5, c2 = 5, log = F)

qbGEV(prob, q_alpha, s_beta, xi, alpha = 0.5, beta = 0.5, p_a = 0.05,
  p_b = 0.2, c1 = 5, c2 = 5)

rbGEV(n, q_alpha, s_beta, xi, alpha = 0.5, beta = 0.5, p_a = 0.05,
  p_b = 0.2, c1 = 5, c2 = 5)
```

Arguments

`y`	scalar quantile.
`q_alpha`	scalar location parameter.
`s_beta`	scalar spread parameter.
`xi`	scalar shape parameter.
`alpha, beta, p_a, p_b, c1, c2`	hyper-parameters for the bGEV distribution, see details. Defaults set to those proposed by Castro-Camilo et al. (2021).
`log`	logical; if `TRUE`, probabilities are given as `log(prob)`.
`prob`	scalar probability.
`n`	number of replications.

Details

The GEV distribution function for real location μ and scale $\sigma > 0$ is

$$G(y|\mu,\sigma,\xi) = \exp[-1\{1+\xi(y-\mu)/\sigma\}_+^{-1/\xi}]$$

for $\xi > 0$ and

$$G(y|\mu,\sigma,\xi) = \exp\{-\exp(-(y-\mu)/\sigma)\}$$

for $\xi = 0$, where $\{x\}_+ = \max\{0, x\}$. It can be re-parametrised in terms of a location parameter q_α for $\alpha \in (0,1)$, denoting the GEV α-quantile, and a spread parameter $s_\beta = q_{1-\beta/2} - q_{\beta/2}$ for $\beta \in (0,1)$. This is achieved using the following one-to-one mapping; if $\xi > 0$, then

$$\mu = q_\alpha - s_\beta(l_{\alpha,\xi} - 1)/(l_{1-\beta/2,\xi} - l_{\beta/2,\xi})$$

and

$$\sigma = \xi s_\beta/(l_{1-\beta/2,\xi} - l_{\beta/2,\xi})$$

where $l_{x,\xi} = (-\log(x))^{-\xi}$; if $\xi = 0$, then

$$\mu = q_\alpha + s_\beta l_\alpha/(l_{\beta/2} - l_{1-\beta/2})$$

and

$$\sigma = s_\beta/(l_{\beta/2} - l_{1-\beta/2})$$

where $l_x = \log(-\log(x))$.

By Castro-Camilo et al. (2021), the blended-GEV has distribution function

$$F(y|q_\alpha, s_\beta, \xi, a, b) = G(y|\tilde{q}_\alpha, \tilde{s}_\beta, \xi = 0)^{1-p(y;a,b)} G(y|_\alpha, s_\beta, \xi)^{p(y;a,b)},$$

for real q_α, $s_\beta > 0$ and $\xi > 0$. The weight function p is defined by $p(y; a, b) = F_{beta}((y - a)/(b - a)|c_1, c_2)$, the distribution function of a beta random variable with shape parameters $c_1 > 3, c_2 > 3$. For continuity of G, we set $a = G^{-1}(p_a|q_\alpha, s_\beta, \xi)$ and $b = G^{-1}(p_b|q_\alpha, s_\beta, \xi)$ for small $0 < p_a < p_b < 1$ and let $\tilde{q}_\alpha = a - (b - a)(l_\alpha - l_{p_a})/(l_{p_a} - l_{p_b})$ and $\tilde{s}_\beta = (b - a)(l_{\beta/2} - l_{1-\beta/2})/(l_{p_a} - l_{p_b})$.

Value

`pbGEV` gives the distribution function; `qbGEV` gives the quantile function; `rbGEV` generates random deviates.

References

Castro-Camilo, D., Huser, R., and Rue, H. (2021), *Practical strategies for generalized extreme value-based regression models for extremes*, Environmetrics, e274. (doi)

`bGEV.NN` *blended-GEV PINN*

Description

Build and train a partially-interpretable neural network for fitting a bGEV model

Usage

```
bGEV.NN.train(Y.train, Y.valid = NULL, X.train.q, X.train.s, type = "MLP",
  link.loc = "identity", n.ep = 100, batch.size = 100, init.loc = NULL,
  init.spread = NULL, init.xi = NULL, widths = c(6, 3),
  filter.dim = c(3, 3), seed = NULL, init.wb_path = NULL, alpha = 0.5,
  beta = 0.5, p_a = 0.05, p_b = 0.2, c1 = 5, c2 = 5,
  S_lambda = NULL)

bGEV.NN.predict(X.train.q, X.train.s, model)
```

Arguments

`Y.train, Y.valid`
: a 2 or 3 dimensional array of training or validation real response values. Missing values can be handled by setting corresponding entries to `Y.train` or `Y.valid` to `-1e5`. The first dimension should be the observation indices, e.g., time.

  If `type=="CNN"`, then `Y.train` and `Y.valid` must have three dimensions with the latter two corresponding to an M by N regular grid of spatial locations. If `Y.valid==NULL`, no validation loss will be computed and the returned model will be that which minimises the training loss over `n.ep` epochs.

`X.train.q`
: list of arrays corresponding to complementary subsets of the $d \geq 1$ predictors which are used for modelling the location parameter q_α. Must contain at least one of the following three named entries:

  `X.train.lin.q` A 3 or 4 dimensional array of "linear" predictor values. One more dimension than `Y.train`. If `NULL`, a model without the linear component is built and trained. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $l_1 \geq 0$ 'linear' predictor values.

  `X.train.add.basis.q` A 4 or 5 dimensional array of basis function evaluations for the "additive" predictor values. The first 2/3 dimensions should be equal to that of `Y.train`; the penultimate dimensions corresponds to the chosen $a_1 \geq 0$ 'linear' predictor values and the last dimension is equal to the number of knots used for estimating the splines. See example. If `NULL`, a model without the additive component is built and trained.

  `X.train.nn.q` A 3 or 4 dimensional array of "non-additive" predictor values. If `NULL`, a model without the NN component is built and trained; if this is the case, then `type` has no effect. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $d-l_1-a_1 \geq 0$ 'non-additive' predictor values.

  Note that `X.train.q` and `X.train.s` are the predictors for both `Y.train` and `Y.valid`.

`X.train.s`
: similarly to `X.train.s`, but for modelling the scale parameter $s_\beta > 0$. Note that both q_β and s_β must be modelled as non-stationary in this version.

`type`
: string defining the type of network to be built. If `type=="MLP"`, the network will have all densely connected layers; if `type=="CNN"`, the network will have all convolutional layers. Defaults to an MLP. (Currently the same network is used for all parameters, may change in future versions)

`n.ep`
: number of epochs used for training. Defaults to 1000.

`batch.size`
: batch size for stochastic gradient descent. If larger than `dim(Y.train)[1]`, i.e., the number of observations, then regular gradient descent used.

`init.loc, init.spread, init.xi`
: sets the initial q_α, s_β and $\xi \in (0,1)$ estimates across all dimensions of `Y.train`. Overridden by `init.wb_path` if `!is.null(init.wb_path)`, but otherwise the initial parameters must be supplied.

`widths`
: vector of widths/filters for hidden dense/convolution layers. Number of layers is equal to `length(widths)`. Defaults to (6,3).

`filter.dim`	if `type=="CNN"`, this 2-vector gives the dimensions of the convolution filter kernel; must have odd integer inputs. Note that filter.dim=c(1,1) is equivalent to `type=="MLP"`. The same filter is applied for each hidden layer across all parameters with NN predictors.
`seed`	seed for random initial weights and biases.
`init.wb_path`	filepath to a `keras` model which is then used as initial weights and biases for training the new model. The original model must have the exact same architecture and trained with the same input data as the new model. If `NULL`, then initial weights and biases are random (with seed `seed`) but the final layer has zero initial weights to ensure that the initial location, spread and shape estimates are `init.loc`, `init.spread` and `init.xi`, respectively, across all dimensions.
`alpha, beta, p_a, p_b`	hyper-parameters associated with the bGEV distribution. Defaults to those used by Castro-Camilo, D., et al. (2021). Require `alpha` >= `p_b` and `beta/2` >= `p_b`.
`S_lambda`	List of smoothing penalty matrices for the splines modelling the effects of `X.train.add.basis.q` and `X.train.add.basis.s` on their respective parameters; each element only used if `!is.null(X.train.add.basis.q)` and `!is.null(X.train.add.basis.s)`, respectively. If `is.null(S_lambda[[1]])`, then no smoothing penalty used for `!is.null(X.train.add.basis.q)`; similarly for the second element and `!is.null(X.train.add.basis.s)`.
`model`	fitted `keras` model. Output from `bGEVPP.NN.train`.
`loc.link`	string defining the link function used for the location parameter, see h_1 below. If `link=="exp"`, then $h_1 = \exp(x)$; if `link=="identity"`, then $h_1(x) = x$.

Details

Consider a real-valued random variable Y and let $\mathbf{X}$ denote a d-dimensional predictor set with observations $\mathbf{x}$. For $i = 1, 2$, we define integers $l_i \geq 0, a_i \geq 0$ and $0 \leq l_i + a_i \leq d$, and let $\mathbf{X}^{(i)}_L, \mathbf{X}^{(i)}_A$ and $\mathbf{X}^{(i)}_N$ be distinct sub-vectors of $\mathbf{X}$, with observations of each component denoted $\mathbf{x}^{(i)}_L, \mathbf{x}^{(i)}_A$ and $\mathbf{x}^{(i)}_N$, respectively; the lengths of the sub-vectors are l_i, a_i and $d_i - l_i - a$, respectively. For a fixed threshold $u(\mathbf{x})$, dependent on predictors, we model $Y|\mathbf{X} = \mathbf{x} \sim$ bGEV$(q_\alpha(\mathbf{x}), s_\beta(\mathbf{x}), \xi)$ for $\xi \in (0, 1)$ with

$$q_\alpha(\mathbf{x}) = h_1[\eta^{(1)}_0 + m^{(1)}_L\{\mathbf{x}^{(1)}_L\} + m^{(1)}_A\{x^{(1)}_A\} + m^{(1)}_N\{\mathbf{x}^{(1)}_N\}]$$

and

$$s_\beta(\mathbf{x}) = \exp[\eta^{(2)}_0 + m^{(2)}_L\{\mathbf{x}^{(2)}_L\} + m^{(2)}_A\{x^{(2)}_A\} + m^{(2)}_N\{\mathbf{x}^{(2)}_N\}]$$

where h_1 is some link-function and $\eta^{(1)}_0, \eta^{(2)}_0$ are constant intercepts. The unknown functions $m^{(1)}_L, m^{(2)}_L$ and $m^{(1)}_A, m^{(2)}_A$ are estimated using linear functions and splines, respectively, and are both returned as outputs by `bGEV.NN.predict`; $m^{(1)}_N, m^{(2)}_N$ are estimated using neural networks (currently the same architecture is used for both parameters). Note that $\xi > 0$ is fixed across all predictors; this may change in future versions.

For details of the bGEV distribution, see `help(pbGEV)`.

The model is fitted by minimising the negative log-likelihood associated with the bGEV model plus some smoothing penalty for the additive functions (determined by `S_lambda`; see Richards and

Huser, 2022); training is performed over `n.ep` training epochs. Although the model is trained by minimising the loss evaluated for `Y.train`, the final returned model may minimise some other loss. The current state of the model is saved after each epoch, using `keras::callback_model_checkpoint`, if the value of some criterion subcedes that of the model from the previous checkpoint; this criterion is the loss evaluated for validation set `Y.valid` if `!is.null(Y.valid)` and for `Y.train`, otherwise.

Value

`bGEV.NN.train` returns the fitted `model`. `bGEV.NN.predict` is a wrapper for `keras::predict` that returns the predicted parameter estimates, and, if applicable, their corresponding linear regression coefficients and spline bases weights.

References

Castro-Camilo, D., Huser, R., and Rue, H. (2021), *Practical strategies for generalized extreme value-based regression models for extremes*, Environmetrics, e274. (doi)

Richards, J. and Huser, R. (2022), *A unifying partially-interpretable framework for neural network-based extreme quantile regression.*

Examples

```
# Build and train a simple MLP for toy data

# Create  predictors
preds<-rnorm(128000)

#Re-shape to a 4d array. First dimension corresponds to observations,
#last to the different components of the predictor set
dim(preds)=c(200,8,8,10) #We have ten predictors

#Split predictors into linear, additive and nn. Different for the location and scale parameters.
X.train.nn.q=preds[,,,1:5] #Five nn predictors for q_\alpha
X.train.lin.q=preds[,,,6:7] #Two linear predictors for q_\alpha
X.train.add.q=preds[,,,8:10] #Three additive predictors for q_\alpha

X.train.nn.s=preds[,,,1:3] #Three nn predictors for s_\beta
X.train.lin.s=preds[,,,4:8] #Five linear predictors for s_\beta
X.train.add.s=preds[,,,9:10] #Two additive predictors for s_\beta


# Create toy response data

#Contribution to location parameter
#Linear contribution
m_L_1 = 0.3*X.train.lin.q[,,,1]+0.6*X.train.lin.q[,,,2]

# Additive contribution
m_A_1 = 0.1*X.train.add.q[,,,1]^3+0.2*X.train.add.q[,,,1]-
  0.1*X.train.add.q[,,,2]^3+0.5*X.train.add.q[,,,2]^2-0.2*X.train.add.q[,,,3]^3

#Non-additive contribution - to be estimated by NN
```

```
m_N_1 = exp(-3+X.train.nn.q[,,,2]+X.train.nn.q[,,,3])+
  sin(X.train.nn.q[,,,1]-X.train.nn.q[,,,2])*(X.train.nn.q[,,,1]+X.train.nn.q[,,,2])-
  cos(X.train.nn.q[,,,3]-X.train.nn.q[,,,4])*(X.train.nn.q[,,,2]+X.train.nn.q[,,,5])

q_alpha=1+m_L_1+m_A_1+m_N_1 #Identity link

#Contribution to scale parameter
#Linear contribution
m_L_2 = 0.2*X.train.lin.s[,,,1]+0.6*X.train.lin.s[,,,2]+0.1*X.train.lin.s[,,,3]-
  0.2*X.train.lin.s[,,,4]+0.5*X.train.lin.s[,,,5]

# Additive contribution
m_A_2 = 0.1*X.train.add.s[,,,1]^2+0.2*X.train.add.s[,,,1]-0.2*X.train.add.s[,,,2]^2+
  0.1*X.train.add.s[,,,2]^3

#Non-additive contribution - to be estimated by NN
m_N_2 = 0.25*exp(-3+X.train.nn.s[,,,2]+X.train.nn.s[,,,3])+
sin(X.train.nn.s[,,,1]-X.train.nn.s[,,,2])*(X.train.nn.s[,,,1]+X.train.nn.s[,,,2])

s_beta=0.3*exp(-2+m_L_2+m_A_2+m_N_2) #Exponential link

xi=0.1 # Set xi

theta=array(dim=c(dim(s_beta),3))
theta[,,,1]=q_alpha; theta[,,,2] = s_beta; theta[,,,3]=xi
#We simulate data from the bGEV distribution

Y=apply(theta,1:3,function(x) rbGEV(1,q_alpha=x[1],s_beta=x[2],xi=x[3]))

#Create training and validation, respectively.
#We mask 20% of the Y values and use this for validation
#Masked values must be set to -1e5 and are treated as missing whilst training

mask_inds=sample(1:length(Y),size=length(Y)*0.8)

Y.train<-Y.valid<-Y #Create training and validation, respectively.
Y.train[-mask_inds]=-1e5
Y.valid[mask_inds]=-1e5



#To build a model with an additive component, we require an array of evaluations of
#the basis functions for each pre-specified knot and entry to X.train.add.q and X.train.add.s

rad=function(x,c){ #Define a basis function. Here we use the radial bases
  out=abs(x-c)^2*log(abs(x-c))
  out[(x-c)==0]=0
 return(out)
}

n.knot.q = 5; n.knot.s = 4 # set number of knots.
#Must be the same for each additive predictor,
#but can differ between the parameters q_\alpha and s_\beta
```

```
#Get knots for q_\alpha predictors
knots.q=matrix(nrow=dim(X.train.add.q)[4],ncol=n.knot.q)

#We set knots to be equally-spaced marginal quantiles
for( i in 1:dim(X.train.add.q)[4]){
knots.q[i,]=quantile(X.train.add.q[,,,i],probs=seq(0,1,length=n.knot.q))
}
#Evaluate radial basis functions for q_\alpha predictors
X.train.add.basis.q<-array(dim=c(dim(X.train.add.q),n.knot.q))
for( i in 1:dim(X.train.add.q)[4]) {
  for(k in 1:n.knot.q) {
    X.train.add.basis.q[,,,i,k]= rad(x=X.train.add.q[,,,i],c=knots.q[i,k])
    #Evaluate rad at all entries to X.train.add.q and for all knots
  }}

  #'#Create smoothing penalty matrix for the three q_alpha additive functions

# Set smoothness parameters for three functions
 lambda = c(0.5,1,0.2)

S_lambda.q=matrix(0,nrow=n.knot.q*dim(X.train.add.q)[4],ncol=n.knot.q*dim(X.train.add.q)[4])
for(i in 1:dim(X.train.add.q)[4]){
 for(j in 1:n.knot.q){
  for(k in 1:n.knot.q){
   S_lambda.q[(j+(i-1)*n.knot.q),(k+(i-1)*n.knot.q)]=lambda[i]*rad(knots.q[i,j],knots.q[i,k])
  }
}
}

#Get knots for s_\beta predictors
knots.s=matrix(nrow=dim(X.train.add.s)[4],ncol=n.knot.s)
for( i in 1:dim(X.train.add.s)[4]){
 knots.s[i,]=quantile(X.train.add.s[,,,i],probs=seq(0,1,length=n.knot.s))
}

#Evaluate radial basis functions for s_\beta predictors
X.train.add.basis.s<-array(dim=c(dim(X.train.add.s),n.knot.s))
for( i in 1:dim(X.train.add.s)[4]) {
  for(k in 1:n.knot.s) {
    X.train.add.basis.s[,,,i,k]= rad(x=X.train.add.s[,,,i],c=knots.s[i,k])
    #Evaluate rad at all entries to X.train.add.q and for all knots
  }}

#Create smoothing penalty matrix for the two s_beta additive functions

# Set smoothness parameters for three functions
 lambda = c(1,1)

S_lambda.s=matrix(0,nrow=n.knot.s*dim(X.train.add.s)[4],ncol=n.knot.s*dim(X.train.add.s)[4])
for(i in 1:dim(X.train.add.s)[4]){
 for(j in 1:n.knot.s){
  for(k in 1:n.knot.s){
    S_lambda.s[(j+(i-1)*n.knot.s),(k+(i-1)*n.knot.s)]=lambda[i]*rad(knots.s[i,j],knots.s[i,k])
  }
}
}

#Join in one list
S_lambda =list("S_lambda.q"=S_lambda.q, "S_lambda.s"=S_lambda.s)


#lin+GAM+NN models defined for both location and scale parameters
X.train.q=list("X.train.nn.q"=X.train.nn.q, "X.train.lin.q"=X.train.lin.q,
               "X.train.add.basis.q"=X.train.add.basis.q) #Predictors for q_\alpha
X.train.s=list("X.train.nn.s"=X.train.nn.s, "X.train.lin.s"=X.train.lin.s,
               "X.train.add.basis.s"=X.train.add.basis.s) #Predictors for s_\beta


#Fit the bGEV model
model<-bGEV.NN.train(Y.train, Y.valid,X.train.q,X.train.s, type="MLP",link.loc="identity",
                       n.ep=500, batch.size=50,init.loc=2, init.spread=2,init.xi=0.1,
                       widths=c(6,3),seed=1,S_lambda=S_lambda)
out<-bGEV.NN.predict(X.train.q=X.train.q,X.train.s=X.train.s,model)

print("q_alpha linear coefficients: "); print(round(out$lin.coeff_q,2))
print("s_beta linear coefficients: "); print(round(out$lin.coeff_s,2))

#To save model, run
#model %>% save_model_tf("model_bGEV")
#To load model, run
# model  <- load_model_tf("model_bGEV",
#  custom_objects=list(
#    "bgev_loss_alpha__beta__p_a__p_b__c1__c2__S_lambda___S_lambda_"=
#        bgev_loss(S_lambda=S_lambda))
#        )

#Note that bGEV_loss() can take custom alpha,beta, p_a and p_b arguments if defaults not used


# Plot splines for the additive predictors

#Location predictors
n.add.preds_q=dim(X.train.add.q)[length(dim(X.train.add.q))]
par(mfrow=c(1,n.add.preds_q))
for(i in 1:n.add.preds_q){
 plt.x=seq(from=min(knots.q[i,]),to=max(knots.q[i,]),length=1000)  #Create sequence for x-axis

  tmp=matrix(nrow=length(plt.x),ncol=n.knot.q)
  for(j in 1:n.knot.q){
    tmp[,j]=rad(plt.x,knots.q[i,j]) #Evaluate radial basis function of plt.x and all knots
  }
  plt.y=tmp%*%out$gam.weights_q[i,]
 plot(plt.x,plt.y,type="l",main=paste0("q_alpha spline: predictor ",i),xlab="x",ylab="f(x)")
  points(knots.q[i,],rep(mean(plt.y),n.knot.q),col="red",pch=2)
  #Adds red triangles that denote knot locations
```

```
}

#Spread predictors
n.add.preds_s=dim(X.train.add.s)[length(dim(X.train.add.s))]
par(mfrow=c(1,n.add.preds_s))
for(i in 1:n.add.preds_s){
 plt.x=seq(from=min(knots.s[i,]),to=max(knots.s[i,]),length=1000)  #Create sequence for x-axis

  tmp=matrix(nrow=length(plt.x),ncol=n.knot.s)
  for(j in 1:n.knot.s){
    tmp[,j]=rad(plt.x,knots.s[i,j]) #Evaluate radial basis function of plt.x and all knots
  }
  plt.y=tmp%*%out$gam.weights_s[i,]
 plot(plt.x,plt.y,type="l",main=paste0("s_beta spline: predictor ",i),xlab="x",ylab="f(x)")
  points(knots.s[i,],rep(mean(plt.y),n.knot.s),col="red",pch=2)
  #Adds red triangles that denote knot locations

}
```

bGEVPP.NN *blended-GEV point process PINN*

Description

Build and train a partially-interpretable neural network for fitting a bGEV point-process model

Usage

```
bGEVPP.NN.train(Y.train, Y.valid = NULL, X.train.q, X.train.s,
  u.train = NULL, type = "MLP", link.loc = "identity", n.ep = 100,
  batch.size = 100, init.loc = NULL, init.spread = NULL,
  init.xi = NULL, widths = c(6, 3), filter.dim = c(3, 3), seed = NULL,
  init.wb_path = NULL, alpha = 0.5, beta = 0.5, p_a = 0.05,
  p_b = 0.2, c1 = 5, c2 = 5, n_b = 1, S_lambda = NULL)

bGEVPP.NN.predict(X.train.q, X.train.s, u.train, model)
```

Arguments

`Y.train, Y.valid`
: a 2 or 3 dimensional array of training or validation real response values. Missing values can be handled by setting corresponding entries to `Y.train` or `Y.valid` to `-1e5`. The first dimension should be the observation indices, e.g., time.

  If `type=="CNN"`, then `Y.train` and `Y.valid` must have three dimensions with the latter two corresponding to an M by N regular grid of spatial locations. If `Y.valid==NULL`, no validation loss will be computed and the returned model will be that which minimises the training loss over `n.ep` epochs.

`X.train.q` list of arrays corresponding to complementary subsets of the $d \geq 1$ predictors which are used for modelling the location parameter q_α. Must contain at least one of the following three named entries:

- `X.train.lin.q` A 3 or 4 dimensional array of "linear" predictor values. One more dimension than `Y.train`. If `NULL`, a model without the linear component is built and trained. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $l_1 \geq 0$ 'linear' predictor values.
- `X.train.add.basis.q` A 4 or 5 dimensional array of basis function evaluations for the "additive" predictor values. The first 2/3 dimensions should be equal to that of `Y.train`; the penultimate dimensions corresponds to the chosen $a_1 \geq 0$ 'linear' predictor values and the last dimension is equal to the number of knots used for estimating the splines. See example. If `NULL`, a model without the additive component is built and trained.
- `X.train.nn.q` A 3 or 4 dimensional array of "non-additive" predictor values. If `NULL`, a model without the NN component is built and trained; if this is the case, then `type` has no effect. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $d-l_1-a_1 \geq 0$ 'non-additive' predictor values.

Note that `X.train.q` and `X.train.s` are the predictors for both `Y.train` and `Y.valid`.

`X.train.s` similarly to `X.train.s`, but for modelling the scale parameter $s_\beta > 0$. Note that both q_β and s_β must be modelled as non-stationary in the current version.

`u.train` an array with the same dimension as `Y.train`. Gives the threshold above which the bGEV-PP model is fitted, see below. Note that `u.train` is applied to both `Y.train` and `Y.valid`.

`type` string defining the type of network to be built. If `type=="MLP"`, the network will have all densely connected layers; if `type=="CNN"`, the network will have all convolutional layers. Defaults to an MLP. (Currently the same network is used for all parameters, may change in future versions)

`n.ep` number of epochs used for training. Defaults to 1000.

`batch.size` batch size for stochastic gradient descent. If larger than `dim(Y.train)[1]`, i.e., the number of observations, then regular gradient descent used.

`init.loc, init.spread, init.xi`
sets the initial q_α, s_β and $\xi \in (0,1)$ estimates across all dimensions of `Y.train`. Overridden by `init.wb_path` if `!is.null(init.wb_path)`, but otherwise the initial parameters must be supplied.

`widths` vector of widths/filters for hidden dense/convolution layers. Number of layers is equal to `length(widths)`. Defaults to (6,3).

`filter.dim` if `type=="CNN"`, this 2-vector gives the dimensions of the convolution filter kernel; must have odd integer inputs. Note that filter.dim=c(1,1) is equivalent to `type=="MLP"`. The same filter is applied for each hidden layer across all parameters with NN predictors.

`seed` seed for random initial weights and biases.

`init.wb_path`	filepath to a `keras` model which is then used as initial weights and biases for training the new model. The original model must have the exact same architecture and trained with the same input data as the new model. If `NULL`, then initial weights and biases are random (with seed `seed`) but the final layer has zero initial weights to ensure that the initial location, spread and shape estimates are `init.loc`, `init.spread` and `init.xi`, respectively, across all dimensions.
`alpha, beta, p_a, p_b, c1, c2`	hyper-parameters associated with the bGEV distribution. Defaults to those used by Castro-Camilo, D., et al. (2021). Require `alpha >= p_b` and `beta/2 >= p_b`.
`n_b`	number of observations per block, e.g., if observations correspond to months and the interest is annual maxima, then `n_b=12`.
`S_lambda`	List of smoothing penalty matrices for the splines modelling the effects of `X.train.add.basis.q` and `X.train.add.basis.s` on their respective parameters; each element only used if `!is.null(X.train.add.basis.q)` and `!is.null(X.train.add.basis.s)`, respectively. If `is.null(S_lambda[[1]])`, then no smoothing penalty used for `!is.null(X.train.add.basis.q)`; similarly for the second element and `!is.null(X.train.add.basis.s)`.
`model`	fitted `keras` model. Output from `bGEVPP.NN.train`.
`loc.link`	string defining the link function used for the location parameter, see h_1 below. If `link=="exp"`, then $h_1 = \exp(x)$; if `link=="identity"`, then $h_1(x) = x$.

Details

Consider a real-valued random variable Y and let $\mathbf{X}$ denote a d-dimensional predictor set with observations $\mathbf{x}$. For $i = 1, 2$, we define integers $l_i \geq 0, a_i \geq 0$ and $0 \leq l_i + a_i \leq d$, and let $\mathbf{X}^{(i)}_L, \mathbf{X}^{(i)}_A$ and $\mathbf{X}^{(i)}_N$ be distinct sub-vectors of $\mathbf{X}$, with observations of each component denoted $\mathbf{x}^{(i)}_L, \mathbf{x}^{(i)}_A$ and $\mathbf{x}^{(i)}_N$, respectively; the lengths of the sub-vectors are l_i, a_i and $d_i - l_i - a$, respectively. For a fixed threshold $u(\mathbf{x})$, dependent on predictors, we model $Y|\mathbf{X} = \mathbf{x} \sim$ bGEV-PP$(q_\alpha(\mathbf{x}), s_\beta(\mathbf{x}), \xi; u(\mathbf{x}))$ for $\xi \in (0, 1)$ with

$$q_\alpha(\mathbf{x}) = h_1[\eta^{(1)}_0 + m^{(1)}_L\{\mathbf{x}^{(1)}_L\} + m^{(1)}_A\{x^{(1)}_A\} + m^{(1)}_N\{\mathbf{x}^{(1)}_N\}]$$

and

$$s_\beta(\mathbf{x}) = \exp[\eta^{(2)}_0 + m^{(2)}_L\{\mathbf{x}^{(2)}_L\} + m^{(2)}_A\{x^{(2)}_A\} + m^{(2)}_N\{\mathbf{x}^{(2)}_N\}]$$

where h_1 is some link-function and $\eta^{(1)}_0, \eta^{(2)}_0$ are constant intercepts. The unknown functions $m^{(1)}_L, m^{(2)}_L$ and $m^{(1)}_A, m^{(2)}_A$ are estimated using linear functions and splines, respectively, and are both returned as outputs by `bGEVPP.NN.predict`; $m^{(1)}_N, m^{(2)}_N$ are estimated using neural networks (currently the same architecture is used for both parameters). Note that $\xi > 0$ is fixed across all predictors; this may change in future versions.

Note that for sufficiently large u that $Y \sim$ bGEV-PP$(q_\alpha, s_\beta, \xi; u)$ implies that $\max_{i=1,\dots,n_b}\{Y_i\} \sim$ bGEV(q_α, s_β, ξ), i.e., the n_b-block maxima of independent realisations of Y follow a bGEV distribution (see `help(pbGEV)`). The size of the block can be specified by the parameter `n_b`.

The model is fitted by minimising the negative log-likelihood associated with the bGEV-PP model plus some smoothing penalty for the additive functions (determined by `S_lambda`; see Richards and Huser, 2022); training is performed over `n.ep` training epochs. Although the model is trained by minimising the loss evaluated for `Y.train`, the final returned model may minimise some other loss. The current state of the model is saved after each epoch, using `keras::callback_model_checkpoint`, if the value of some criterion subcedes that of the model from the previous checkpoint; this criterion is the loss evaluated for validation set `Y.valid` if `!is.null(Y.valid)` and for `Y.train`, otherwise.

Value

`bGEVPP.NN.train` returns the fitted `model`. `bGEVPP.NN.predict` is a wrapper for `keras::predict` that returns the predicted parameter estimates, and, if applicable, their corresponding linear regression coefficients and spline bases weights.

References

Castro-Camilo, D., Huser, R., and Rue, H. (2021), *Practical strategies for generalized extreme value-based regression models for extremes*, Environmetrics, e274. (doi)

Richards, J. and Huser, R. (2022), *A unifying partially-interpretable framework for neural network-based extreme quantile regression.*

Examples

```
# Build and train a simple MLP for toy data

# Create  predictors
preds<-rnorm(128000)

#Re-shape to a 4d array. First dimension corresponds to observations,
#last to the different components of the predictor set
dim(preds)=c(200,8,8,10) #We have ten predictors

#Split predictors into linear, additive and nn. Different for the location and scale parameters.
X.train.nn.q=preds[,,,1:5] #Five nn predictors for q_\alpha
X.train.lin.q=preds[,,,6:7] #Two linear predictors for q_\alpha
X.train.add.q=preds[,,,8:10] #Three additive predictors for q_\alpha

X.train.nn.s=preds[,,,1:3] #Three nn predictors for s_\beta
X.train.lin.s=preds[,,,4:8] #Five linear predictors for s_\beta
X.train.add.s=preds[,,,9:10] #Two additive predictors for s_\beta


# Create toy response data

#Contribution to location parameter
#Linear contribution
m_L_1 = 0.3*X.train.lin.q[,,,1]+0.6*X.train.lin.q[,,,2]

# Additive contribution
m_A_1 = 0.1*X.train.add.q[,,,1]^3+0.2*X.train.add.q[,,,1]-
  0.1*X.train.add.q[,,,2]^3+0.5*X.train.add.q[,,,2]^2-0.2*X.train.add.q[,,,3]^3

#Non-additive contribution - to be estimated by NN
m_N_1 = exp(-3+X.train.nn.q[,,,2]+X.train.nn.q[,,,3])+
  sin(X.train.nn.q[,,,1]-X.train.nn.q[,,,2])*(X.train.nn.q[,,,1]+X.train.nn.q[,,,2])-
  cos(X.train.nn.q[,,,3]-X.train.nn.q[,,,4])*(X.train.nn.q[,,,2]+X.train.nn.q[,,,5])

q_alpha=1+m_L_1+m_A_1+m_N_1 #Identity link

#Contribution to scale parameter
#Linear contribution
m_L_2 = 0.2*X.train.lin.s[,,,1]+0.6*X.train.lin.s[,,,2]+0.1*X.train.lin.s[,,,3]-
  0.2*X.train.lin.s[,,,4]+0.5*X.train.lin.s[,,,5]

# Additive contribution
m_A_2 = 0.1*X.train.add.s[,,,1]^2+0.2*X.train.add.s[,,,1]-0.2*X.train.add.s[,,,2]^2+
  0.1*X.train.add.s[,,,2]^3

#Non-additive contribution - to be estimated by NN
m_N_2 = exp(-3+X.train.nn.s[,,,2]+X.train.nn.s[,,,3])+
  sin(X.train.nn.s[,,,1]-X.train.nn.s[,,,2])*(X.train.nn.s[,,,1]+X.train.nn.s[,,,2])

s_beta=0.5*exp(-2+m_L_2+m_A_2+m_N_2) #Exponential link

xi=0.1 # Set xi

theta=array(dim=c(dim(s_beta),3))
theta[,,,1]=q_alpha; theta[,,,2] = s_beta; theta[,,,3]=xi
#We simulate data from the extreme value point process model with u take as the 80% quantile

#Gives the 80% quantile of Y
u<-apply(theta,1:3,function(x) qPP(prob=0.8,loc=x[1],scale=x[2],xi=x[3],re.par = T))

#Simulate from re-parametrised point process model using same u as given above
Y=apply(theta,1:3,function(x) rPP(1,u.prob=0.8,loc=x[1],scale=x[2],xi=x[3],re.par=T))

# Note that the point process model is only valid for Y > u. If Y < u, then rPP gives NA.
# We can set NA values to some c < u as these do not contribute to model fitting.
Y[is.na(Y)]=u[is.na(Y)]-1



#Create training and validation, respectively.
#We mask 20% of the Y values and use this for validation
#Masked values must be set to -1e5 and are treated as missing whilst training

mask_inds=sample(1:length(Y),size=length(Y)*0.8)

Y.train<-Y.valid<-Y #Create training and validation, respectively.
Y.train[-mask_inds]=-1e5
Y.valid[mask_inds]=-1e5



#To build a model with an additive component, we require an array of evaluations of
#the basis functions for each pre-specified knot and entry to X.train.add.q and X.train.add.s
```

```
rad=function(x,c){ #Define a basis function. Here we use the radial bases
  out=abs(x-c)^2*log(abs(x-c))
  out[(x-c)==0]=0
 return(out)
}

n.knot.q = 5; n.knot.s = 4 # set number of knots.
#Must be the same for each additive predictor,
#but can differ between the parameters q_\alpha and s_\beta

#Get knots for q_\alpha predictors
knots.q=matrix(nrow=dim(X.train.add.q)[4],ncol=n.knot.q)

#We set knots to be equally-spaced marginal quantiles
for( i in 1:dim(X.train.add.q)[4]){
 knots.q[i,]=quantile(X.train.add.q[,,,i],probs=seq(0,1,length=n.knot.q))
}

#Evaluate radial basis functions for q_\alpha predictors
X.train.add.basis.q<-array(dim=c(dim(X.train.add.q),n.knot.q))
for( i in 1:dim(X.train.add.q)[4]) {
  for(k in 1:n.knot.q) {
    X.train.add.basis.q[,,,i,k]= rad(x=X.train.add.q[,,,i],c=knots.q[i,k])
    #Evaluate rad at all entries to X.train.add.q and for all knots
  }}

#Create smoothing penalty matrix for the three q_alpha additive functions

# Set smoothness parameters for three functions
 lambda = c(0.5,1,0.2)

S_lambda.q=matrix(0,nrow=n.knot.q*dim(X.train.add.q)[4],ncol=n.knot.q*dim(X.train.add.q)[4])
for(i in 1:dim(X.train.add.q)[4]){
 for(j in 1:n.knot.q){
  for(k in 1:n.knot.q){
   S_lambda.q[(j+(i-1)*n.knot.q),(k+(i-1)*n.knot.q)]=lambda[i]*rad(knots.q[i,j],knots.q[i,k])
  }
}
}


#Get knots for s_\beta predictors
knots.s=matrix(nrow=dim(X.train.add.s)[4],ncol=n.knot.s)
for( i in 1:dim(X.train.add.s)[4]){
 knots.s[i,]=quantile(X.train.add.s[,,,i],probs=seq(0,1,length=n.knot.s))
}

#Evaluate radial basis functions for s_\beta predictors
X.train.add.basis.s<-array(dim=c(dim(X.train.add.s),n.knot.s))
for( i in 1:dim(X.train.add.s)[4]) {
  for(k in 1:n.knot.s) {
    X.train.add.basis.s[,,,i,k]= rad(x=X.train.add.s[,,,i],c=knots.s[i,k])
    #Evaluate rad at all entries to X.train.add.q and for all knots
  }}

#Create smoothing penalty matrix for the two s_beta additive functions

# Set smoothness parameters for three functions
 lambda = c(1,1)

S_lambda.s=matrix(0,nrow=n.knot.s*dim(X.train.add.s)[4],ncol=n.knot.s*dim(X.train.add.s)[4])
for(i in 1:dim(X.train.add.s)[4]){
 for(j in 1:n.knot.s){
  for(k in 1:n.knot.s){
   S_lambda.s[(j+(i-1)*n.knot.s),(k+(i-1)*n.knot.s)]=lambda[i]*rad(knots.s[i,j],knots.s[i,k])
  }
}
}

#Join in one list
S_lambda =list("S_lambda.q"=S_lambda.q, "S_lambda.s"=S_lambda.s)

#lin+GAM+NN models defined for both location and scale parameters
X.train.q=list("X.train.nn.q"=X.train.nn.q, "X.train.lin.q"=X.train.lin.q,
               "X.train.add.basis.q"=X.train.add.basis.q) #Predictors for q_\alpha
X.train.s=list("X.train.nn.s"=X.train.nn.s, "X.train.lin.s"=X.train.lin.s,
               "X.train.add.basis.s"=X.train.add.basis.s) #Predictors for s_\beta

#We treat u as fixed and known. In an application, u can be estimated using quant.NN.train.

u.train <- u

#Fit the bGEV-PP model using u.train
model<-bGEVPP.NN.train(Y.train, Y.valid,X.train.q,X.train.s, u.train, type="MLP",link.loc="identity",
                       n.ep=500, batch.size=50,init.loc=2, init.spread=2,init.xi=0.1,
                       widths=c(6,3),seed=1, n_b=12,S_lambda=S_lambda)
out<-bGEVPP.NN.predict(X.train.q=X.train.q,X.train.s=X.train.s,u.train=u.train,model)

print("q_alpha linear coefficients: "); print(round(out$lin.coeff_q,2))
print("s_beta linear coefficients: "); print(round(out$lin.coeff_s,2))

#To save model, run
#model %>% save_model_tf("model_bGEVPP")
#To load model, run
# model  <- load_model_tf("model_bGEVPP",
#  custom_objects=list(
#    "bgev_PP_loss_alpha__beta__p_a__p_b__c1__c2__n_b__S_lambda___S_lambda_"=
#      bgev_PP_loss(n_b=12,S_lambda=S_lambda))
#        )

#Note that bGEV_PP_loss() can take custom alpha,beta, p_a, p_b, c1 and c2 arguments if defaults not used.


# Plot splines for the additive predictors

#Location predictors
```

```
n.add.preds_q=dim(X.train.add.q)[length(dim(X.train.add.q))]
par(mfrow=c(1,n.add.preds_q))
for(i in 1:n.add.preds_q){
  plt.x=seq(from=min(knots.q[i,]),to=max(knots.q[i,]),length=1000)  #Create sequence for x-axis

   tmp=matrix(nrow=length(plt.x),ncol=n.knot.q)
   for(j in 1:n.knot.q){
     tmp[,j]=rad(plt.x,knots.q[i,j]) #Evaluate radial basis function of plt.x and all knots
   }
   plt.y=tmp%*%out$gam.weights_q[i,]
  plot(plt.x,plt.y,type="l",main=paste0("q_alpha spline: predictor ",i),xlab="x",ylab="f(x)")
   points(knots.q[i,],rep(mean(plt.y),n.knot.q),col="red",pch=2)
   #Adds red triangles that denote knot locations

}

#Spread predictors
n.add.preds_s=dim(X.train.add.s)[length(dim(X.train.add.s))]
par(mfrow=c(1,n.add.preds_s))
for(i in 1:n.add.preds_s){
  plt.x=seq(from=min(knots.s[i,]),to=max(knots.s[i,]),length=1000)  #Create sequence for x-axis

   tmp=matrix(nrow=length(plt.x),ncol=n.knot.s)
   for(j in 1:n.knot.s){
     tmp[,j]=rad(plt.x,knots.s[i,j]) #Evaluate radial basis function of plt.x and all knots
   }
   plt.y=tmp%*%out$gam.weights_s[i,]
  plot(plt.x,plt.y,type="l",main=paste0("s_beta spline: predictor ",i),xlab="x",ylab="f(x)")
   points(knots.s[i,],rep(mean(plt.y),n.knot.s),col="red",pch=2)
   #Adds red triangles that denote knot locations

}
```

evPP *The extreme value point process*

Description

Quantile function and random generation for the extreme value point process with location equal to `loc`, scale equal to `scale` and shape equal to `xi >= 0`. Note that unlike similar functions in package `stats`, these functions accept only scalar inputs, rather than vectors, for the parameters.

Usage

```
qPP(prob, loc, scale, xi, n_b = 1, re.par = F, alpha = 0.5, beta = 0.5,
  tol = 1e-04, qMax = 10000)

rPP(n, u.prob, loc, scale, xi, n_b = 1, re.par = F, alpha = 0.5,
  beta = 0.5, tol = 1e-04, qMax = 10000)
```

Arguments

`prob`	scalar probability.
`loc`	location parameter. If `re.par==FALSE`, then `loc` corresponds to μ; otherwise, `loc` corresponds to q_α.
`scale`	scale parameter. If `re.par==FALSE`, then `scale` corresponds to σ; otherwise, `scale` corresponds to s_β.
`xi`	shape parameter. Require `xi >= 0`.
`n_b`	number of observations per block, e.g., if observations correspond to months and the interest is annual maxima, then `n_b=12`. See details.
`re.par`	logical; if `TRUE`, then the corresponding GEV used the alternative parameterisation.
`alpha, beta`	hyper-parameters for the reparameterisation, see details. Defaults set both to `0.5`. Only used if `re.par==TRUE`.
`tol`	tolerance for the numerical solver. Defaults to `1e-4`.
`qMax`	finite upper and lower bounds used by the numerical solver. If the absolute value of the output from `qPP` or maximum output from `rPP` is close to `qMax`, then `qMax` needs increasing at the cost of computation time. Defaults to `1e4`.
`n`	number of replications.
`u.prob`	exceedance probability for threshold u.

Details

Following Coles (2001), consider a sequence of independent random variables $Y_1, \ldots, Y_n$ with common distribution function F. For n_b-block-maxima $M_{n_b} = \max\{Y_1, \ldots, Y_{n_b}\}$; if there exists sequences $\{a_n > 0\}$ and $\{b_n\}$ such that

$$\Pr\{(M_{n_b} - b_n)/a_n \le z\} \to G(z) \;\text{ as }\; n_b \to \infty,$$

for non-degenerate G, then G is the generalised extreme value $\text{GEV}(\mu, \sigma, \xi)$ distribution function, see `help{pbGEV}`. If $\xi > 0$, then G has finite lower-endpoint $z_- = \mu - \sigma/\xi$; if $\xi = 0$, then the lower-endpoint is infinite.

Assume that the above limit holds and $\xi \ge 0$. Then for any $u > z_-$, the sequence of point processes $N_n = \{(i/(n+1), (Y_i - b_n)/a_n) : i = 1, \ldots, n\}$ converges on regions $(0,1) \times (u, \infty)$ as $n \to \infty$ to a Poisson point process with intensity measure Λ of the form $\Lambda(A) = -(n/n_b)(t_2 - t_1)\log G(z)$, where $A = [t_1, t_2] \times [z, \infty)$ for $0 \le t_1 \le t_2 \le 1$. We consider unit inter-arrival times and so set $t_2 - t_1 = 1$. Here the functions `qPP` and `rPP` give the quantile function and random generation of Y assuming that the Poisson process limit holds for Y above u. The threshold u is taken to be the `u.prob` quantile of Y.

Castro-Camilo et al. (2021) propose a reparameterisation of the GEV distribution in terms of a location parameter q_α for $\alpha \in (0,1)$, denoting the GEV α-quantile, and a spread parameter $s_\beta = q_{1-\beta/2} - q_{\beta/2}$ for $\beta \in (0,1)$; for the full mapping, see `help{pbGEV}`. If `re.par==TRUE`, then the input `loc` and `scale` correspond to q_α and s_β, rather than μ and σ.

Distribution function inversion is performed numerically using the bisection method.

Value

qPP gives the quantile function and rPP generates n random deviates. Any simulated values subceding the threshold u are treated as censored and set to NA.

References

Coles, S. G. (2001), *An Introduction to Statistical Modeling of Extreme Values*. Volume 208, Springer. (doi)

Castro-Camilo, D., Huser, R., and Rue, H. (2021), *Practical strategies for generalized extreme value-based regression models for extremes*, Environmetrics, e274. (doi)

Examples

```
set.seed(1)
loc<-3; scale<-4; xi<-0.2 #Parameter values

u<-qPP(prob=0.9,loc,scale,xi) #Gives the 90% quantile of Y

#Create 1000 realisations of Y with exceedance threshold equal to u.
#Note that the input to rPP is the exceedance probability u.prob, not the threshold itself
Y<-rPP(1000,u.prob=0.9,loc,scale,xi)
hist(Y)
#Note that values Y<u are censored and set to NA
```

GPD.NN *GPD PINN*

Description

Build and train a partially-interpretable neural network for fitting a GPD model

Usage

```
GPD.NN.train(Y.train, Y.valid = NULL, X.train, u.train = NULL,
  type = "MLP", link = "identity", tau = NULL, n.ep = 100,
  batch.size = 100, init.scale = NULL, init.xi = NULL, widths = c(6,
  3), filter.dim = c(3, 3), seed = NULL, init.wb_path = NULL,
  S_lambda = NULL)

GPD.NN.predict(X.train, u.train, model)
```

Arguments

`Y.train, Y.valid`
: a 2 or 3 dimensional array of training or validation real response values. Missing values can be handled by setting corresponding entries to `Y.train` or `Y.valid` to `-1e5`. The first dimension should be the observation indices, e.g., time.

  If `type=="CNN"`, then `Y.train` and `Y.valid` must have three dimensions with the latter two corresponding to an M by N regular grid of spatial locations. If `Y.valid==NULL`, no validation loss will be computed and the returned model will be that which minimises the training loss over `n.ep` epochs.

`X.train`
: list of arrays corresponding to complementary subsets of the $d \geq 1$ predictors which are used for modelling the scale parameter σ. Must contain at least one of the following three named entries:

  `X.train.lin` A 3 or 4 dimensional array of "linear" predictor values. One more dimension then `Y.train`. If `NULL`, a model without the linear component is built and trained. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $l \geq 0$ 'linear' predictor values.

  `X.train.add.basis` A 4 or 5 dimensional array of basis function evaluations for the "additive" predictor values. The first 2/3 dimensions should be equal to that of `Y.train`; the penultimate dimensions corresponds to the chosen $a \geq 0$ 'linear' predictor values and the last dimension is equal to the number of knots used for estimating the splines. See example. If `NULL`, a model without the additive component is built and trained.

  `X.train.nn` A 3 or 4 dimensional array of "non-additive" predictor values. If `NULL`, a model without the NN component is built and trained; if this is the case, then `type` has no effect. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $d-l-a \geq 0$ 'non-additive' predictor values.

  Note that `X.train` is the predictors for both `Y.train` and `Y.valid`.

`u.train`
: an array with the same dimension as `Y.train`. Gives the threshold used to create exceedances of `Y.train`, see below. Note that `u.train` is applied to both `Y.train` and `Y.valid`.

`type`
: string defining the type of network to be built. If `type=="MLP"`, the network will have all densely connected layers; if `type=="CNN"`, the network will have all convolutional layers. Defaults to an MLP. (Currently the same network is used for all parameters, may change in future versions)

`n.ep`
: number of epochs used for training. Defaults to 1000.

`batch.size`
: batch size for stochastic gradient descent. If larger than `dim(Y.train)[1]`, i.e., the number of observations, then regular gradient descent used.

`init.scale, init.xi`
: sets the initial $sigma$ and $\xi \in (0,1)$ estimates across all dimensions of `Y.train`. Overridden by `init.wb_path` if `!is.null(init.wb_path)`, but otherwise the initial parameters must be supplied.

`widths`
: vector of widths/filters for hidden dense/convolution layers. Number of layers is equal to `length(widths)`. Defaults to (6,3).

`filter.dim`	if `type=="CNN"`, this 2-vector gives the dimensions of the convolution filter kernel; must have odd integer inputs. Note that filter.dim=c(1,1) is equivalent to `type=="MLP"`. The same filter is applied for each hidden layer across all parameters with NN predictors.
`seed`	seed for random initial weights and biases.
`init.wb_path`	filepath to a `keras` model which is then used as initial weights and biases for training the new model. The original model must have the exact same architecture and trained with the same input data as the new model. If `NULL`, then initial weights and biases are random (with seed `seed`) but the final layer has zero initial weights to ensure that the initial scale and shape estimates are `init.scale` and `init.xi`, respectively, across all dimensions.
`model`	fitted `keras` model. Output from `GPD.NN.train`.
`S_lamda`	smoothing penalty matrix for the splines modelling the effect of `X.train.add.basis` on $\log(\sigma)$; only used if `!is.null(X.train.add.basis)`. If `is.null(S_lambda)`, then no smoothing penalty used.

Details

Consider a real-valued random variable Y and let $\mathbf{X}$ denote a d-dimensional predictor set with observations $\mathbf{x}$. For integers $l \geq 0, a \geq 0$ and $0 \leq l + a \leq d$, let $\mathbf{X}_L, \mathbf{X}_A$ and $\mathbf{X}_N$ be distinct sub-vectors of $\mathbf{X}$, with observations of each component denoted $\mathbf{x}_L, \mathbf{x}_A$ and $\mathbf{x}_N$, respectively; the lengths of the sub-vectors are l, a and $d - l - a$, respectively. For a fixed threshold $u(\mathbf{x})$, dependent on predictors, we model $(Y - u(\mathbf{x}))|\mathbf{X} = \mathbf{x} \sim \text{GPD}(\sigma(\mathbf{X})), \xi; u(\mathbf{x}))$ for $\xi \in (0, 1)$ with

$$\sigma(\mathbf{x}) = \exp[\eta_0 + m_L\{\mathbf{x}_L\} + m_A\{x_A\} + m_N\{\mathbf{x}_N\}]$$

where η_0 is a constant intercept. The unknown functions m_L and m_A are estimated using linear functions and splines, respectively, and are both returned as outputs by `GPD.NN.predict`; m_N is estimated using a neural network (currently the same architecture is used for both parameters). Note that $\xi > 0$ is fixed across all predictors; this may change in future versions.

The model is fitted by minimising the negative log-likelihood associated with the GPD model over `n.ep` training epochs. Although the model is trained by minimising the loss evaluated for `Y.train`, the final returned model may minimise some other loss. The current state of the model is saved after each epoch, using `keras::callback_model_checkpoint`, if the value of some criterion subcedes that of the model from the previous checkpoint; this criterion is the loss evaluated for validation set `Y.valid` if `!is.null(Y.valid)` and for `Y.train`, otherwise.

Value

`bGEVPP.NN.train` returns the fitted `model`. `bGEVPP.NN.predict` is a wrapper for `keras::predict` that returns the predicted parameter estimates, and, if applicable, their corresponding linear regression coefficients and spline bases weights.

References

Coles, S. (2001), *An Introduction to Statistical Modeling of Extreme Values*, Springer Series in Statistics. (doi)

Richards, J. and Huser, R. (2022), *A unifying partially-interpretable framework for neural network-based extreme quantile regression.*

Examples

```
#Apply model to toy data

# Create  predictors
preds<-rnorm(128000)

#Re-shape to a 4d array. First dimension corresponds to observations,
#last to the different components of the predictor set
dim(preds)=c(200,8,8,10) #We have ten predictors

#Split predictors into linear, additive and nn.

X.train.nn=preds[,,,1:5] #Five nn predictors
X.train.lin=preds[,,,6:8] #Three linear predictors
X.train.add=preds[,,,9:10] #Two additive predictors


# Create response data

#Contribution to scale parameter
#Linear contribution
m_L = 0.5*X.train.lin[,,,1]+0.3*X.train.lin[,,,2]-0.4*X.train.lin[,,,3]

# Additive contribution
m_A = 0.2*X.train.add[,,,1]^2+0.05*X.train.add[,,,1]-0.1*X.train.add[,,,2]^2+
  0.01*X.train.add[,,,2]^3

#Non-additive contribution - to be estimated by NN
m_N =0.5*(exp(-4+X.train.nn[,,,2]+X.train.nn[,,,3])+
            sin(X.train.nn[,,,1]-X.train.nn[,,,2])*(X.train.nn[,,,1]+X.train.nn[,,,2])-
            cos(X.train.nn[,,,3]-X.train.nn[,,,4])*(X.train.nn[,,,2]+X.train.nn[,,,5]))

sigma=2*exp(-2+m_L+m_A+m_N) #Exponential link
xi=0.1 # Set xi

#We simulate data as exceedances above some random positive threshold u.
u<-apply(sigma,1:3,function(x) rgpd(n=1,loc=0,scale=1,shape=0.1) ) #Random threshold

theta=array(dim=c(dim(sigma),3))
theta[,,,1]=sigma; theta[,,,2]  =xi; theta[,,,3]  =u

#If u were the true 80% quantile, say,  of the response, then only 20% of the data should exceed u.
#We achieve this by simulating a Bernoulli variable to determine if Y exceeds u

Y=apply(theta,1:3,function(x){
  if(rbinom(1,1,0.8)==1) rgpd(n=1,loc=x[3],scale=x[1],shape=x[2]) else  runif(1,0,x[3])
}
) #Simulate GPD exceedances above u as given above


#Create training and validation, respectively.
#We mask 20% of the Y values and use this for validation
```

```
#Masked values must be set to -1e5 and are treated as missing whilst training

mask_inds=sample(1:length(Y),size=length(Y)*0.8)

Y.train<-Y.valid<-Y #Create training and validation, respectively.
Y.train[-mask_inds]=-1e5
Y.valid[mask_inds]=-1e5



#To build a model with an additive component, we require an array of evaluations of
#the basis functions for each pre-specified knot and entry to X.train.add

rad=function(x,c){ #Define a basis function. Here we use the radial bases
  out=abs(x-c)^2*log(abs(x-c))
  out[(x-c)==0]=0
  return(out)
}

n.knot = 5 # set number of knots. Must be the same for each additive predictor

knots=matrix(nrow=dim(X.train.add)[4],ncol=n.knot)

#We set knots to be equally-spaced marginal quantiles
for( i in 1:dim(X.train.add)[4]){
 knots[i,]=quantile(X.train.add[,,,i],probs=seq(0,1,length=n.knot))
}

X.train.add.basis<-array(dim=c(dim(X.train.add),n.knot))
for( i in 1:dim(X.train.add)[4]) {
  for(k in 1:n.knot) {
    X.train.add.basis[,,,i,k]= rad(x=X.train.add[,,,i],c=knots[i,k])
    #Evaluate rad at all entries to X.train.add and for all knots
  }}

#Create smoothing penalty matrix for the two additive functions

# Set smoothness parameters for three functions
 lambda = c(0.5,1)

S_lambda=matrix(0,nrow=n.knot*dim(X.train.add)[4],ncol=n.knot*dim(X.train.add)[4])
for(i in 1:dim(X.train.add)[4]){
 for(j in 1:n.knot){
  for(k in 1:n.knot){
     S_lambda[(j+(i-1)*n.knot),(k+(i-1)*n.knot)]=lambda[i]*rad(knots[i,j],knots[i,k])
  }
}
}

#lin+GAM+NN models defined for scale parameter
X.train=list("X.train.nn"=X.train.nn, "X.train.lin"=X.train.lin,
             "X.train.add.basis"=X.train.add.basis)
```

```
#We treat u as fixed and known. In an application, u can be estimated using quant.NN.train.

u.train <- u

#Fit the GPD model for exceedances above u.train
model<-GPD.NN.train(Y.train, Y.valid,X.train, u.train, type="MLP",
                    n.ep=500, batch.size=50,init.scale=1, init.xi=0.1,
                    widths=c(6,3),seed=1,S_lambda=S_lambda)
out<-GPD.NN.predict(X.train=X.train,u.train=u.train,model)

print("sigma linear coefficients: "); print(round(out$lin.coeff_sigma,2))

#To save model, run
#model %>% save_model_tf("model_GPD")
To load model, run
#model  <- load_model_tf("model_GPD",
#custom_objects=list("GPD_loss_S_lambda___S_lambda_"=GPD_loss(S_lambda=S_lambda)))


# Plot splines for the additive predictors
n.add.preds=dim(X.train.add)[length(dim(X.train.add))]
par(mfrow=c(1,n.add.preds))
for(i in 1:n.add.preds){
 plt.x=seq(from=min(knots[i,]),to=max(knots[i,]),length=1000)  #Create sequence for x-axis

  tmp=matrix(nrow=length(plt.x),ncol=n.knot)
  for(j in 1:n.knot){
    tmp[,j]=rad(plt.x,knots[i,j]) #Evaluate radial basis function of plt.x and all knots
  }
  plt.y=tmp%*%out$gam.weights_sigma[i,]
 plot(plt.x,plt.y,type="l",main=paste0("sigma spline: predictor ",i),xlab="x",ylab="f(x)")
  points(knots[i,],rep(mean(plt.y),n.knot),col="red",pch=2)
  #Adds red triangles that denote knot locations

}
```

logistic.NN *Logistic regression PINN*

Description

Build and train a partially-interpretable neural network for a logistic regression model

Usage

```
logistic.NN.train(Y.train, Y.valid = NULL, X.train, type = "MLP",
  n.ep = 100, batch.size = 100, init.p = NULL, widths = c(6, 3),
  filter.dim = c(3, 3), seed = NULL, init.wb_path = NULL,
  S_lambda = NULL)
```

```
logistic.NN.predict(X.train, model)
```

Arguments

`Y.train, Y.valid`	a 2 or 3 dimensional array of training or validation response values, with entries of 0/1 for failure/success. Missing values can be handled by setting corresponding entries to `Y.train` or `Y.valid` to `-1e5`. The first dimension should be the observation indices, e.g., time. If `type=="CNN"`, then `Y.train` and `Y.valid` must have three dimensions with the latter two corresponding to an M by N regular grid of spatial locations. If `Y.valid==NULL`, no validation loss will be computed and the returned model will be that which minimises the training loss over `n.ep` epochs.
`X.train`	list of arrays corresponding to complementary subsets of the $d \geq 1$ predictors which are used for modelling. Must contain at least one of the following three named entries: `X.train.lin` A 3 or 4 dimensional array of "linear" predictor values. One more dimension than `Y.train`. If `NULL`, a model without the linear component is built and trained. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $l \geq 0$ 'linear' predictor values. `X.train.add.basis` A 4 or 5 dimensional array of basis function evaluations for the "additive" predictor values. The first 2/3 dimensions should be equal to that of `Y.train`; the penultimate dimensions corresponds to the chosen $a \geq 0$ 'linear' predictor values and the last dimension is equal to the number of knots used for estimating the splines. See example. If `NULL`, a model without the additive component is built and trained. `X.train.nn` A 3 or 4 dimensional array of "non-additive" predictor values. If `NULL`, a model without the NN component is built and trained; if this is the case, then `type` has no efect. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $d-l-a \geq 0$ 'non-additive' predictor values. Note that `X.train` is the predictors for both `Y.train` and `Y.valid`.
`type`	string defining the type of network to be built. If `type=="MLP"`, the network will have all densely connected layers; if `type=="CNN"`, the network will have all convolutional layers. Defaults to an MLP.
`n.ep`	number of epochs used for training. Defaults to 1000.
`batch.size`	batch size for stochastic gradient descent. If larger than `dim(Y.train)[1]`, i.e., the number of observations, then regular gradient descent used.
`init.p`	sets the initial probability estimate across all dimensions of `Y.train`. Defaults to empirical estimate. Overriden by `init.wb_path` if `!is.null(init.wb_path)`.
`widths`	vector of widths/filters for hidden dense/convolution layers. Number of layers is equal to `length(widths)`. Defaults to (6,3).
`filter.dim`	if `type=="CNN"`, this 2-vector gives the dimensions of the convolution filter kernel; must have odd integer inputs. Note that filter.dim=c(1,1) is equivalent to `type=="MLP"`. The same filter is applied for each hidden layer.

`seed`	seed for random initial weights and biases.
`init.wb_path`	filepath to a `keras` model which is then used as initial weights and biases for training the new model. The original model must have the exact same architecture and trained with the same input data as the new model. If `NULL`, then initial weights and biases are random (with seed `seed`) but the final layer has zero initial weights to ensure that the initial probability estimate is `init.p` across all dimensions.
`model`	fitted `keras` model. Output from `logistic.NN.train`.
`S_lamda`	smoothing penalty matrix for the splines modelling the effect of `X.train.add.basis` on logit(p); only used if `!is.null(X.train.add.basis)`. If `is.null(S_lambda)`, then no smoothing penalty used.

Details

Consider a Bernoulli random variable, say $Z \sim$ Bernoulli(p), with probability mass function $\Pr(Z = 1) = p = 1 - \Pr(Z = 0) = 1 - p$. Let $Y \in \{0, 1\}$ be a univariate Boolean response and let $\mathbf{X}$ denote a d-dimensional predictor set with observations $\mathbf{x}$. For integers $l \geq 0, a \geq 0$ and $0 \leq l + a \leq d$, let $\mathbf{X}_L, \mathbf{X}_A$ and $\mathbf{X}_N$ be distinct sub-vectors of $\mathbf{X}$, with observations of each component denoted $\mathbf{x}_L, \mathbf{x}_A$ and $\mathbf{x}_N$, respectively; the lengths of the sub-vectors are l, a and $d - l - a$, respectively. We model $Y|\mathbf{X} = \mathbf{x} \sim$ Bernoulli$(p(\mathbf{x}))$ with

$$p(\mathbf{x}) = h[\eta_0 + m_L\{\mathbf{x}_L\} + m_A\{x_A\} + m_N\{\mathbf{x}_N\}],$$

where h is the logistic link-function and η_0 is a constant intercept. The unknown functions m_L and m_A are estimated using a linear function and spline, respectively, and are both returned as outputs by `logistic.NN.predict`; m_N is estimated using a neural network.

The model is fitted by minimising the binary cross-entropy loss plus some smoothing penalty for the additive functions (determined by `S_lambda`; see Richards and Huser, 2022) over `n.ep` training epochs. Although the model is trained by minimising the loss evaluated for `Y.train`, the final returned model may minimise some other loss. The current state of the model is saved after each epoch, using `keras::callback_model_checkpoint`, if the value of some criterion subcedes that of the model from the previous checkpoint; this criterion is the loss evaluated for validation set `Y.valid` if `!is.null(Y.valid)` and for `Y.train`, otherwise.

Value

`logistic.NN.train` returns the fitted `model`. `logistic.NN.predict` is a wrapper for `keras::predict` that returns the predicted probability estimates, and, if applicable, the linear regression coefficients and spline bases weights.

Examples

```
# Build and train a simple MLP for toy data

# Create 'nn', 'additive' and 'linear' predictors
X.train.nn<-rnorm(5000); X.train.add<-rnorm(2000); X.train.lin<-rnorm(3000)
```

```
#Re-shape to a 4d array. First dimension corresponds to observations,
#last to the different components of the predictor set
dim(X.train.nn)=c(10,10,10,5) #Five nn predictors
dim(X.train.lin)=c(10,10,10,3) #Three linear predictors
dim(X.train.add)=c(10,10,10,2) #Two additive predictors

# Create toy response data

#Linear contribution
m_L = 0.3*X.train.lin[,,,1]+0.6*X.train.lin[,,,2]-0.2*X.train.lin[,,,3]

# Additive contribution
m_A = 0.1*X.train.add[,,,1]^2+0.2*X.train.add[,,,1]-0.1*X.train.add[,,,2]^3+
0.5*X.train.add[,,,2]^2

#Non-additive contribution - to be estimated by NN
m_N = exp(-3+X.train.nn[,,,2]+X.train.nn[,,,3])+
sin(X.train.nn[,,,1]-X.train.nn[,,,2])*(X.train.nn[,,,4]+X.train.nn[,,,5])

p=0.5+0.5*tanh((m_L+m_A+m_N)/2) #Logistic link
Y=apply(p,1:3,function(x) rbinom(1,1,x))

#Create training and validation, respectively.
#We mask 20% of the Y values and use this for validation
#Masked values must be set to -1e5 and are treated as missing whilst training

mask_inds=sample(1:length(Y),size=length(Y)*0.8)

Y.train<-Y.valid<-Y #Create training and validation, respectively.
Y.train[-mask_inds]=-1e5
Y.valid[mask_inds]=-1e5



#To build a model with an additive component, we require an array of evaluations of
#the basis functions for each pre-specified knot and entry to X.train.add

rad=function(x,c){ #Define a basis function. Here we use the radial bases
  out=abs(x-c)^2*log(abs(x-c))
  out[(x-c)==0]=0
  return(out)
}

n.knot = 5 # set number of knots. Must be the same for each additive predictor
knots=matrix(nrow=dim(X.train.add)[4],ncol=n.knot)

#We set knots to be equally-spaced marginal quantiles
for( i in 1:dim(X.train.add)[4]) {
knots[i,]=quantile(X.train.add[,,,i],probs=seq(0,1,length=n.knot))}

X.train.add.basis<-array(dim=c(dim(X.train.add),n.knot))
for( i in 1:dim(X.train.add)[4]) {
for(k in 1:n.knot) {
```

```
X.train.add.basis[,,,i,k]= rad(x=X.train.add[,,,i],c=knots[i,k])
#Evaluate rad at all entries to X.train.add and for all knots
}}

 #Penalty matrix for additive functions

# Set smoothness parameters for first and second additive functions
 lambda = c(0.5,1)

S_lambda=matrix(0,nrow=n.knot*dim(X.train.add)[4],ncol=n.knot*dim(X.train.add)[4])
for(i in 1:dim(X.train.add)[4]){
 for(j in 1:n.knot){
  for(k in 1:n.knot){
     S_lambda[(j+(i-1)*n.knot),(k+(i-1)*n.knot)]=lambda[i]*rad(knots[i,j],knots[i,k])
  }
}
}

X.train=list("X.train.nn"=X.train.nn, "X.train.lin"=X.train.lin,
"X.train.add.basis"=X.train.add.basis)

#Build and train a two-layered "lin+GAM+NN" MLP
model<-logistic.NN.train(Y.train, Y.valid,X.train,  type="MLP",n.ep=2000,
                      batch.size=50,init.p=0.4, widths=c(6,3),
                      S_lambda=S_lambda)

out<-predict_bernoulli_nn(X.train,model)
hist(out$predictions) #Plot histogram of predicted probability
print(out$lin.coeff)

n.add.preds=dim(X.train.add)[length(dim(X.train.add))]
par(mfrow=c(1,n.add.preds))
for(i in 1:n.add.preds){
 plt.x=seq(from=min(knots[i,]),to=max(knots[i,]),length=1000)  #Create sequence for x-axis

 tmp=matrix(nrow=length(plt.x),ncol=n.knot)
 for(j in 1:n.knot){
   tmp[,j]=rad(plt.x,knots[i,j]) #Evaluate radial basis function of plt.x and all knots
 }
 plt.y=tmp%*%out$gam.weights[i,]
 plot(plt.x,plt.y,type="l",main=paste0("Quantile spline: predictor ",i),xlab="x",ylab="f(x)")
 points(knots[i,],rep(mean(plt.y),n.knot),col="red",pch=2)
 #Adds red triangles that denote knot locations
}

#To save model, run model %>% save_model_tf("model_Bernoulli")
#To load model, run model  <- load_model_tf("model_Bernoulli",
#custom_objects=list("bce_loss_S_lambda___S_lambda_"=bce.loss(S_lambda)))
```

quant.NN *Non-parametric single quantile regression PINN*

Description

Build and train a partially-interpretable neural network for non-parametric single quantile regression

Usage

```
quant.NN.train(Y.train, Y.valid = NULL, X.train, type = "MLP",
  link = "identity", tau = NULL, n.ep = 100, batch.size = 100,
  init.q = NULL, widths = c(6, 3), filter.dim = c(3, 3), seed = NULL,
  init.wb_path = NULL, S_lambda = NULL)

quant.NN.predict(X.train, model)
```

Arguments

`Y.train, Y.valid`
: a 2 or 3 dimensional array of training or validation real response values. Missing values can be handled by setting corresponding entries to `Y.train` or `Y.valid` to `-1e5`. The first dimension should be the observation indices, e.g., time.

  If `type=="CNN"`, then `Y.train` and `Y.valid` must have three dimensions with the latter two corresponding to an M by N regular grid of spatial locations. If `Y.valid==NULL`, no validation loss will be computed and the returned model will be that which minimises the training loss over `n.ep` epochs.

`X.train`
: list of arrays corresponding to complementary subsets of the $d \geq 1$ predictors which are used for modelling. Must contain at least one of the following three named entries:

  `X.train.lin` A 3 or 4 dimensional array of "linear" predictor values. One more dimension than `Y.train`. If `NULL`, a model without the linear component is built and trained. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $l \geq 0$ 'linear' predictor values.

  `X.train.add.basis` A 4 or 5 dimensional array of basis function evaluations for the "additive" predictor values. The first 2/3 dimensions should be equal to that of `Y.train`; the penultimate dimensions corresponds to the chosen $a \geq 0$ 'linear' predictor values and the last dimension is equal to the number of knots used for estimating the splines. See example. If `NULL`, a model without the additive component is built and trained.

  `X.train.nn` A 3 or 4 dimensional array of "non-additive" predictor values. If `NULL`, a model without the NN component is built and trained; if this is the case, then `type` has no effect. The first 2/3 dimensions should be equal to that of `Y.train`; the last dimension corresponds to the chosen $d-l-a \geq 0$ 'non-additive' predictor values.

  Note that `X.train` is the predictors for both `Y.train` and `Y.valid`.

`type`
: string defining the type of network to be built. If `type=="MLP"`, the network will have all densely connected layers; if `type=="CNN"`, the network will have all convolutional layers. Defaults to an MLP.

`link`
: string defining the link function used, see h below. If `link=="exp"`, then $h = \exp(x)$; if `link=="identity"`, then $h(x) = x$.

`tau`	quantile level. Must satisfy `0 < tau < 1`.
`n.ep`	number of epochs used for training. Defaults to 1000.
`batch.size`	batch size for stochastic gradient descent. If larger than `dim(Y.train)[1]`, i.e., the number of observations, then regular gradient descent used.
`init.q`	sets the initial `tau`-quantile estimate across all dimensions of `Y.train`. Defaults to empirical estimate. Overriden by `init.wb_path` if `!is.null(init.wb_path)`.
`widths`	vector of widths/filters for hidden dense/convolution layers. Number of layers is equal to `length(widths)`. Defaults to (6,3).
`filter.dim`	if `type=="CNN"`, this 2-vector gives the dimensions of the convolution filter kernel; must have odd integer inputs. Note that filter.dim=c(1,1) is equivalent to `type=="MLP"`. The same filter is applied for each hidden layer.
`seed`	seed for random initial weights and biases.
`init.wb_path`	filepath to a `keras` model which is then used as initial weights and biases for training the new model. The original model must have the exact same architecture and trained with the same input data as the new model. If `NULL`, then initial weights and biases are random (with seed `seed`) but the final layer has zero initial weights to ensure that the initial quantile estimate is `init.q` across all dimensions.
`model`	fitted `keras` model. Output from `quant.NN.train`.
`S_lamda`	smoothing penalty matrix for the splines modelling the effect of `X.train.add.basis` on the inverse-`link` of the `tau`-quantile; only used if `!is.null(X.train.add.basis)`. If `is.null(S_lambda)`, then no smoothing penalty used.

Details

Consider a real-valued random variable Y and let $\mathbf{X}$ denote a d-dimensional predictor set with observations $\mathbf{x}$. For integers $l \geq 0, a \geq 0$ and $0 \leq l + a \leq d$, let $\mathbf{X}_L, \mathbf{X}_A$ and $\mathbf{X}_N$ be distinct sub-vectors of $\mathbf{X}$, with observations of each component denoted $\mathbf{x}_L, \mathbf{x}_A$ and $\mathbf{x}_N$, respectively; the lengths of the sub-vectors are l, a and $d-l-a$, respectively. We model $\Pr\{Y \leq y_\tau(\mathbf{x}) | \mathbf{X} = \mathbf{x}\} = \tau$ with

$$y_\tau(\mathbf{x}) = h[\eta_0 + m_L\{\mathbf{x}_L\} + m_A\{x_A\} + m_N\{\mathbf{x}_N\}],$$

where h is some link-function and η_0 is a constant intercept. The unknown functions m_L and m_A are estimated using a linear function and spline, respectively, and are both returned as outputs by `quant.NN.predict`; m_N is estimated using a neural network.

The model is fitted by minimising the penalised tilted loss over `n.ep` training epochs; the loss is given by

$$l(y_\tau; y) = \max\{\tau(y - y_\tau), (\tau - 1)(y - y_\tau)\}$$

plus some smoothing penalty for the additive functions (determined by `S_lambda`; see Richards and Huser, 2022) and is averaged over all entries to `Y.train` (or `Y.valid`). Although the model is trained by minimising the loss evaluated for `Y.train`, the final returned model may minimise some other loss. The current state of the model is saved after each epoch, using `keras::callback_model_checkpoint`, if the value of some criterion subcedes that of the model from the previous checkpoint; this criterion is the loss evaluated for validation set `Y.valid` if `!is.null(Y.valid)` and for `Y.train`, otherwise.

Value

quant.NN.train returns the fitted model. quant.NN.predict is a wrapper for keras::predict that returns the predicted tau-quantile estimates, and, if applicable, the linear regression coefficients and spline bases weights.

References

Richards, J. and Huser, R. (2022), *A unifying partially-interpretable framework for neural network-based extreme quantile regression.*

Examples

```
# Build and train a simple MLP for toy data

# Create 'nn', 'additive' and 'linear' predictors
X.train.nn<-rnorm(5000); X.train.add<-rnorm(2000); X.train.lin<-rnorm(3000)

#Re-shape to a 4d array. First dimension corresponds to observations,
#last to the different components of the predictor set
dim(X.train.nn)=c(10,10,10,5) #Five nn predictors
dim(X.train.lin)=c(10,10,10,3) #Three linear predictors
dim(X.train.add)=c(10,10,10,2) #Two additive predictors

# Create toy response data

#Linear contribution
m_L = 0.3*X.train.lin[,,,1]+0.6*X.train.lin[,,,2]-0.2*X.train.lin[,,,3]

# Additive contribution
m_A = 0.1*X.train.add[,,,1]^2+0.2*X.train.add[,,,1]-0.1*X.train.add[,,,2]^3+
0.5*X.train.add[,,,2]^2

#Non-additive contribution - to be estimated by NN
m_N = exp(-3+X.train.nn[,,,2]+X.train.nn[,,,3])+
sin(X.train.nn[,,,1]-X.train.nn[,,,2])*(X.train.nn[,,,4]+X.train.nn[,,,5])

theta=1+m_L+m_A+m_N #Identity link
#We simulate normal data and estimate the median, i.e., the 50% quantile or mean,
#as the form for this is known
Y=apply(theta,1:3,function(x) rnorm(1,mean=x,sd=2))

#Create training and validation, respectively.
#We mask 20% of the Y values and use this for validation.
#Masked values must be set to -1e5 and are treated as missing whilst training

mask_inds=sample(1:length(Y),size=length(Y)*0.8)

Y.train<-Y.valid<-Y #Create training and validation, respectively.
Y.train[-mask_inds]=-1e5
Y.valid[mask_inds]=-1e5
```

```
#To build a model with an additive component, we require an array of evaluations of
#the basis functions for each pre-specified knot and entry to X.train.add

rad=function(x,c){ #Define a basis function. Here we use the radial bases
  out=abs(x-c)^2*log(abs(x-c))
  out[(x-c)==0]=0
  return(out)
}

n.knot = 5 # set number of knots. Must be the same for each additive predictor
knots=matrix(nrow=dim(X.train.add)[4],ncol=n.knot)

#We set knots to be equally-spaced marginal quantiles
for( i in 1:dim(X.train.add)[4]){
 knots[i,]=quantile(X.train.add[,,,i],probs=seq(0,1,length=n.knot))
 }

X.train.add.basis<-array(dim=c(dim(X.train.add),n.knot))
for( i in 1:dim(X.train.add)[4]) {
for(k in 1:n.knot) {
X.train.add.basis[,,,i,k]= rad(x=X.train.add[,,,i],c=knots[i,k])
#Evaluate rad at all entries to X.train.add and for all knots
}}


#Penalty matrix for additive functions

# Set smoothness parameters for first and second additive functions
 lambda = c(0.5,1)

S_lambda=matrix(0,nrow=n.knot*dim(X.train.add)[4],ncol=n.knot*dim(X.train.add)[4])
for(i in 1:dim(X.train.add)[4]){
 for(j in 1:n.knot){
  for(k in 1:n.knot){
     S_lambda[(j+(i-1)*n.knot),(k+(i-1)*n.knot)]=lambda[i]*rad(knots[i,j],knots[i,k])
  }
}
}

#Build lin+GAM+NN model.
X.train=list("X.train.nn"=X.train.nn, "X.train.lin"=X.train.lin,
"X.train.add.basis"=X.train.add.basis)

#Build and train a two-layered "lin+GAM+NN" MLP
model<-quant.NN.train(Y.train, Y.valid,X.train, type="MLP",link="identity",tau=0.5,n.ep=500,
                      batch.size=50, widths=c(6,3),S_lambda=S_lambda)

out<-quant.NN.predict(X.train,model)
hist(out$predictions) #Plot histogram of predicted quantiles
print(out$lin.coeff)
```

```
n.add.preds=dim(X.train.add)[length(dim(X.train.add))]
par(mfrow=c(1,n.add.preds))
for(i in 1:n.add.preds){
 plt.x=seq(from=min(knots[i,]),to=max(knots[i,]),length=1000)  #Create sequence for x-axis

  tmp=matrix(nrow=length(plt.x),ncol=n.knot)
  for(j in 1:n.knot){
    tmp[,j]=rad(plt.x,knots[i,j]) #Evaluate radial basis function of plt.x and all knots
  }
  plt.y=tmp%*%out$gam.weights[i,]
 plot(plt.x,plt.y,type="l",main=paste0("Quantile spline: predictor ",i),xlab="x",ylab="f(x)")
  points(knots[i,],rep(mean(plt.y),n.knot),col="red",pch=2)
  #Adds red triangles that denote knot locations
}

tau <- 0.5
#To save model, run
# model %>% save_model_tf(paste0("model_",tau,"-quantile"))
#To load model, run
#model  <- load_model_tf(paste0("model_",tau,"-quantile"),
#custom_objects=list("tilted_loss_tau___tau__S_lambda_"=tilted.loss(tau,S_lambda)))
```

USwild *US Wildfires data*

Description

Data used by Richards and Huser (2022) for modelling extreme wildfires in the contiguous U.S. with partially-interpretable neural networks

Usage

```
data(USwild)
```

Format

A list with 5 elements:

BA An array with dimension (276, 119, 51), corresponding to the burnt area response data.

X A list with 3 elements:

- **X.met** An array with dimension (276, 119, 51, 10) corresponding to the meterological predictors described below.
- **X.lc** An array with dimension (276, 119, 51, 18) corresponding to the land cover predictors described below.
- **X.oro** An array with dimension (276, 119, 51, 2) corresponding to the oropgraphical predictors described below.

times A vector of length 276 giving the observation indices. Format is "year-month". Corresponds to first dimension of `BA`.

lon A vector of length 119 giving the longitude ordinate for the second dimension of `BA`.

lat A vector of length 51 giving the latitude ordinate for the third dimension of `BA`.

Details

The response data `BA` are observations of monthly aggregated burnt area (acres) of 3503 spatial grid-cells located across the contiguous United States, with the states of Alaska and Hawaii excluded. The observation period covers 1993 to 2015, using only months between March and September, inclusive, leaving 161 observed spatial fields. Grid-cells are arranged on a regular latitude/longitude grid with spatial resolution 0.5deg by 0.5deg. Observations are provided by the Fire Program Analysis fire-occurrence database (Short, 2017) which collates U.S. wildfire records from the reporting systems of federal, state and local organisations.

Both the response data and the subsequently described predictors have been re-gridded to a regular spatio-temporal grid with missing values set to `-1e5`. For `BA` and entries to `X`, the first three dimensions correspond to time $\times$ latitude $\times$ longitude with their respective ordinate values given in `times`, `lat` and `lon`.

We consider three types of predictor variables, given in `X`: these are meteorological (`X.met`), land cover proportions (`X.lc`) and orograhical (`X.oro`).

Ten meteorological variables are considered and given as monthly means in `X.met`. These variables are provided by the ERA5-reanalysis on land surface, available through the COPERNICUS Climate Data Service, which is given on a 0.1deg $\times$ 0.1deg grid; the values are then aggregated to a 0.5deg $\times$ 0.5deg resolution. The variables are ordered as followed: both eastern and northern components of wind velocity at a 10m altitude (m/s), both dew-point temperature and temperature at a 2m altitude (Kelvin), potential evaporation (m), evaporation (m), precipitation (m), surface pressure (Pa) and surface net solar, and thermal, radiation (J/m^2). This particular ERA5-reanalysis samples over land only, and so the meteorological conditions over the oceans are not available.

The land cover variables that are given in `X.lc` describe the proportion of a grid-cell which is covered by one of 18 different types, e.g., urban, grassland, water (see Opitz (2022) for full details). Land cover predictors are derived using a gridded land cover map, of spatial resolution 300m and temporal resolution one year, produced by COPERNICUS and available through their Climate Data Service. For each 0.5deg $\times$ 0.5deg grid-cell, the proportion of a cell consisting of a specific land cover type is derived from the high-resolution product. The variables are ordered lc1 - lc18, as described by Opitz (2022).

The two orographical predictors given in `X.oro` are the mean and standard deviation of the altitude (m) for each grid-cell; estimates are derived using a densely-sampled gridded output from the U.S. Geographical Survey Elevation Point Query Service.

References

Richards, J. and Huser, R. (2022), *High-dimensional extreme quantile regression using partially-interpretable neural networks: With application to U.S. wildfires.*

Short, K. C. (2017), *Spatial wildfire occurrence data for the United States, 1992-2015 FPA_FOD_20170508. 4th Ed. Fort Collins, CO: Forest Service Research Data Archive.*

Opitz, T.. (2022), *Editorial: EVA 2021 Data Competition on spatio-temporal prediction of wildfire activity in the United States.* Extremes, to appear.

Examples

```
data(USwild)
```

Index